LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Design of a Memory Controller for the MIT Tagged Token Dataflow Machine

Computation Structures Group Memo 230
11 October 1983

Steven K. Heller

Arvind

A version of this paper will be presented
at the IEEE / ICCD '83
Portchester, NY
October 31 - November 3, 1983

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0661. The first author was supported by the International Business Machines Corporation as a part of MIT's VI-A program.

Source File = ISC.MSS.85, Last updated 11 October 1983 at 8:21pm

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Table of Contents

List of Figures

Abstract

I-structure storage is a new type of random access storage for multiprocessor systems. By employing extra *status bits*, data slots can be tagged as *data-present* or *data-absent*, allowing reads which precede writes to be deferred until the data arrives. In order to implement an I-structure storage in hardware, a sophisticated controller is required. A functional overview of the I-structure Storage Controller (ISC) is presented. Based on our experience developing the ISC in IDL (Interactive Design Language), we discuss certain hardware design language issues which will challenge the hardware design languages of the future.

Key words and phrases: hardware design languages, IDL, multiprocessors, I-structure storage, storage controller, dataflow

Design of a Memory Controller for the MIT Tagged Token Dataflow Machine

1. Introduction

All multiprocessor machines have to avoid read-before-write and write-write races which result when two processors access the same memory location. Write-write races can be avoided by a suitable programming model, but read-before-write races are usually avoided by sacrificing parallelism in application code. I-structure storage has been proposed to avoid read-before-write races at the basic hardware level [3]. I-structure storage, or simply I-store, constitutes an integral part of the Tagged-token Dataflow Machine [1, 2], being developed in the MIT Laboratory for Computer Science by the Functional Languages and Architecture group. The machine is currently being simulated in Pascal, and the work on emulating a 64 processor version of the machine on 64 Lisp machines (Symbolics 3600's) has begun. Depending upon the success of the emulated machine, a custom VLSI version of the machine may be built after two to three years.

In this paper we will describe I-structure storage operations (Section 2), and our experience in developing a controller for I-structure storage (ISC) in the Interactive Design Language (IDL). The IDL design system [8, 9], developed at IBM, Poughkeepsie, allows a designer to design, simulate, and modify a VLSI design. The system includes facilities for specifying, testing, and manipulating a design as well as a direct path to silicon through PLA personality. The IDL system is particularly well suited for designing Finite State Machines (FSMs). A brief description of IDL is given in Section 3.

It became clear early on that the ISC was too complex to be designed at the logic level directly. Since many important hardware parameters (*e.g.*, RAM speed, controller technology) could not be specified when the design was undertaken, we decided to separate the control logic of the ISC from the rest of the design with the understanding that the control logic (PLA) may have to be suitably modified as other design decisions became firm. Because of our inexperience in hardware design, we underestimated the complexity of the control logic program. The fact that the original specification of the I-store operations was incomplete and in some cases inconsistent also contributed to enlarging the design. We think that one person could not have completed this design without using IDL, and we are convinced that IDL or similar design aids are indispensable if the ISC or the rest of the dataflow machine design is to be committed to silicon.

Specification of the ISC in IDL took approximately 1500 lines of code. The ISC design did, however, stress the IDL system to the limit, and in order to facilitate even larger designs, another generation of behavioral design systems is needed. In Sections 4 and 5 we will describe several design issues which provide a difficult challenge for the hardware design languages of the future. On one hand the level of abstraction in digital design has to be raised if designs of ISC magnitude are to be carried out routinely by one or two designer. On the other hand, greater control over the hardware generated must be provided if the designs are to be of practical value.

2. I-structure Storage and Associated Operations

A data structure known as I-structure has been proposed by Arvind and Thomas to efficiently manipulate arrays in functional languages [5]. I-structure storage and associated operations form an implementation model for I-structures (see [6] for more details).

2.1. I-structures

An I-structure can be viewed as an array of slots, where slots can be filled in any order, and each slot is filled at most once. The array name can be treated as the address of the first slot (*i.e.*, a descriptor), not unlike the name of an array in Fortran. In fact an I-structure descriptor (ISD) also contains information about the length (number of elements), width (size of an element), and type of the structure. However, unlike a Fortran array, the semantics of I-structures permit passing the descriptor of an array to other parts of the code even before all the slots have been filled. (That is, slots that will be filled and are needed for the computation. This feature increases parallelism in a program, especially when the program is decomposed into a producer and a consumer of a data structure.) A read request (*i.e.*, x[i]) to a slot is processed whenever the slot is filled. If the slot is never filled, then the read request never gets satisfied.

There are two kinds of I-structures: *uniform* and *mixed*. Each element of a *uniform* structure has the same type and hence length. Since the type can be easily stored in the ISD, type information does not need to be stored with each entry. Uniform I-structures are stored in a format called *u-fix* (untyped fixed-length); they are written using a **store-u-fix* operation and read using a **fetch-notype-stored* operation. Since no type information is stored with each entry, a type must be provided when reading a structure of uniform type.

In a *mixed* I-structure, each element can have a different type and length. We have developed two formats for storing mixed I-structures. The *t-fix* (typed fixed-length) format allocates to each element as much space as the largest element requires. The *t-var* (typed variable-length) format allocates a minimum amount of space to each element, and if a value doesn't fit, an internal pointer points to a larger slot where the value actually is stored. This use of invisible pointers was inspired by various LISP machines (*e.g.* Symbolics 3600) and is explained more fully in Section 2.3. The type is stored along with each element in the t-fix and t-var formats. The **store-t-fix* and **store-t-var* operations are used to write into t-fix and t-var formats respectively, and both formats can be read using the **fetch-type-stored* operation.

2.2. I-store Implementation

The implementation of an I-store is fundamentally different from that of a conventional memory in two ways. Locations are tagged as *data-present* or *data-not-present*; and if data is not present and a read occurs, the read is *deferred* until the data arrives. The information required in order to monitor the "status" of a location is stored in a hidden *status* field.

In our model, a destination (48 bits) is associated with each **fetch* instruction, and indicates the computation to which the fetched data is to be sent. If a **fetch* precedes the **store* operation for a piece of data, the ISC remembers the destination carried by the **fetch*. The destination is used by the token switching network to route tokens to their "destinations." Other multiprocessor systems that wish to use an I-store might simply combine a processor number with a locally unique

computation identifier to form a destination. A computation identifier is needed since processors should expect to send many requests for data before the first response is received [4]. (The use of 48 bits to name a destination computation may seem excessive, but it is needed in our machine because each instruction constitutes a separate computation and many copies of the same instruction may execute concurrently.)

The ISC will assist storage managers by providing additional status information and accommodating variable length data. While the presence bit and the deferred bit are the only status bits required for a minimal implementation of I-structures, additional status information is used to assure consistency against possible compiler bugs and physical hardware boundaries.

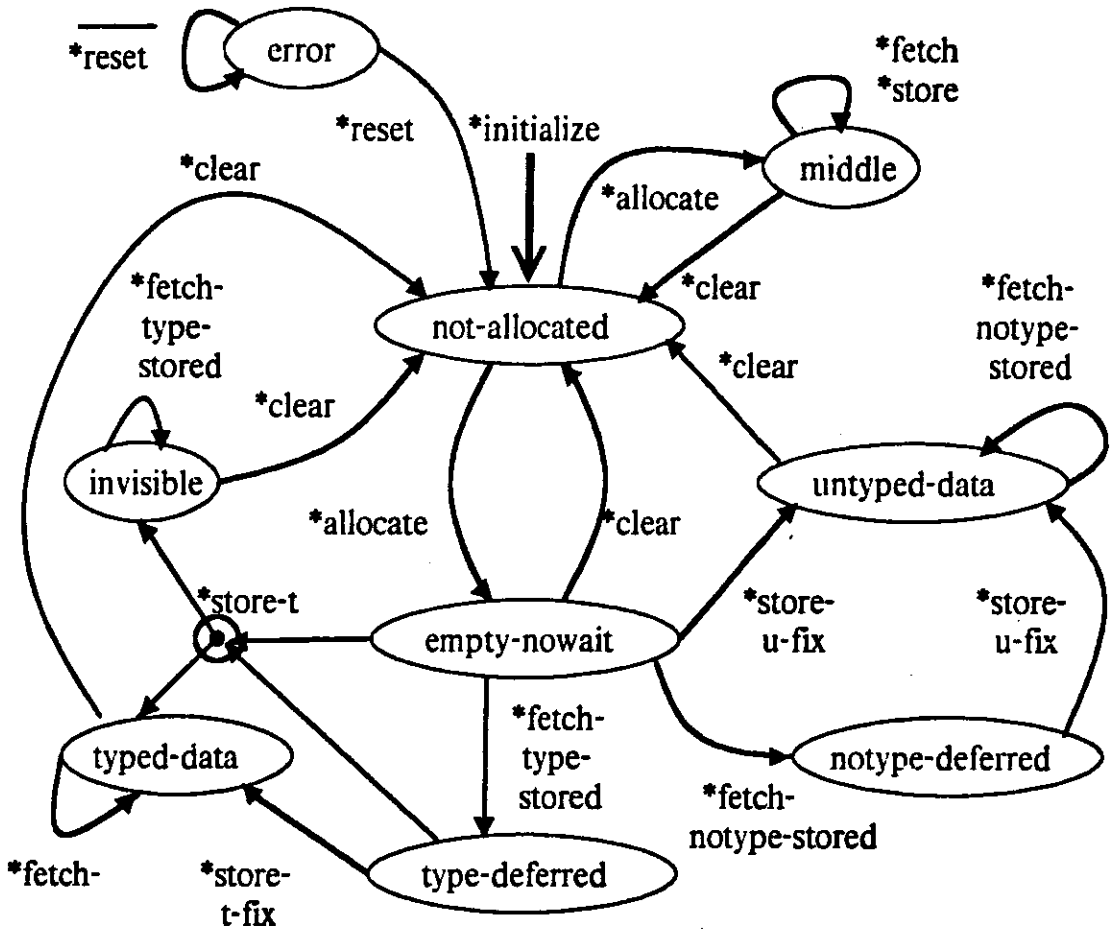
The ISC design assumes 288K bytes of Random Access Memory (RAM), registers, a Finite State Machine (FSM), and some random logic. The I-structure Memory (ISM) is divided between two physical memories: a 256K byte data storage, and a 32K byte status storage. The data storage has a 16 bit address space and a 32 bit wide word. The status storage also has a 16 bit address space, but has a 4 bit wide word. Together they store 64K words of 32 bits each, with 4 bits of hidden status associated with each word. The 64K words are divided into two sections: the *data-section* and the *link-section*. The location at which the data-section ends and the link-section begins is programmable and is set when the system is initialized. Using the **initialize* operation, the link-section is initialized as a linked list of double words (*free-links*). The status field of the second word is used to indicate a cell at the end of the list, and the second half of the second word is used to point to the next free-link. The **initialize* operation also sets the status of each word in the data-section to *not-allocated*. (The address space for the I-store has been extended to 24 bits since this work was completed (January 1982). However all internal pointers (ones pointing to the link section) are still 16 bits. In this way, a much larger address space for data is achieved without changing the internal data structures.)

2.3. Normal I-store Operations

Normal I-store operations can be understood with the help of the status transition diagram for storage cells given in Figure 2-1. For example, suppose an array of n 2-word elements is allocated starting with location 0. This operation will cause the status of words 0,2,4,...,2(n-1) to be changed to *empty-nowait* (see Figures 2-1 and 2-2a). The status of words 1,3,5,...,2n-1 will be changed to *middle*. If the machine and the compiler are working correctly there is no possibility of generating the address of a word whose status is marked as *middle*. When a **store* operation is performed, say on the first element (*i.e.* words 0 and 1) of the array, the status of word 0 is changed to *untyped-data* if the operation is **store-u-fix*, and to *typed-data* if the operation is **store-t-fix* (see Figure 2-2b). The **store-t-var* operation only uses one word from the data-section. If a **store-t-var* operation attempts to store data which is longer than one word, an *invisible pointer* is stored in the word. The invisible pointer points to the data which is actually stored in the link-section. The status of the word in the data-section is set to *invisible*.

If a **fetch* operation follows the corresponding **store*, the data structure remains intact, and the data is sent to the destination provided by the **fetch*. If the **fetch* precedes the **store*, however, the destination is remembered in a cell from the free list, a pointer to the cell is left behind, and the status bits are set to *deferred* (see Figure 2-2c). When the data arrives on the **store* operation, the data is sent to all deferred destinations, the storage occupied by the deferred destinations is put back on the free list, and the status of the data word is changed to either *untyped-data*, *typed-data*, or

invisible as shown in Figure 2-2b.



Notes:
1) All omitted arrows are branches to error
2) The circled dot is a deterministic decision based on both the operation and the length of the data

Figure 2-1: Status Transitions in the Data-Section

Thus normal operations on I-store can be summarized as follows. The **initialize* operation initializes the ISM using two parameters: the boundary between the data-section and the link-section, and the size at which the free list should spawn a warning message. The **allocate* operation is used to prepare an array of locations for stores and fetches. The **clear* operation is used to recover a sequence of locations, and return them to the unallocated state. The **store-t-fix*, **store-t-var*, and **store-u-fix* operations are used to store data of types t-fix, t-var, and u-fix respectively. The **fetch-type-stored* operation retrieves data of type t-fix or t-var (data stored along with its type). The **fetch-notype-stored* operation retrieves data of type u-fix (data stored without its type). The **reset* operation is used to recover a word from an error state. The **allocate-free-space* operation increases the size of the link-section (and the free list) by allocating a

space in the data-section for free cells; this space can be returned only by reinitializing the entire ISM.

2.3.1. Error Handling in the I-store Controller

Since the ISC operates at a fairly high level, it is possible to detect certain errors. There are three types of errors that can occur in the ISC: *status errors*, *data type errors*, and *free list errors*. A *status error* indicates that the operation being performed is inconsistent with the status of the addressed word; an attempt to overwrite causes a status error. *Data type* errors indicate an inconsistency in the structure of the data being operated on; data being forced into a slot which is too small to contain it is a data type error. *Free list* errors indicate that the free list is low or out of free cells. In our analysis, we assumed the existence of an *error manager* which can be thought of as a piece of software that deals with errors and inconsistencies as they occur. If an error occurs at any location in the I-store, the high order bit of a status field is set to one, the lower order bits are left unchanged, and information is sent to the error manager. Information is thereby preserved about the structure of a data element that must be untangled by the error manager.

The operations needed by the error and storage managers are summarized below. The **read-absolute* and **write-absolute* operations fetch and store a word directly; the **load-error-destination* operation loads the contents of a six byte register, called the error destination register, which contains the address of the manager to which all errors are sent; and the **get-free-size* operation sends the contents of the *free-size* register to the specified destination. The true size of the free list can be determined given the value stored in *free-size* and a constant (provided the last time the ISM was initialized).

3. Introduction to the IDL Design System

IDL, developed by Maissel and Ostapko [8], is the hardware design environment in which the ISC is implemented. This Section provides an overview of the IDL system, and a brief description of the base language.

3.1. Overview of the IDL Design System

The IDL design system allows a designer to explore, design, simulate, verify, document, and modify a VLSI design. The system includes an interactive user interface, a file manager, a language (IDL, a register transfer level language), a compiler, an assembler, several simulators, minimization programs, and several other logic manipulation tools. The system outputs both a canonical (two level boolean) form of the logic designed and a self documenting specification of the design. The IDL system is particularly well suited for designing Finite State Machines (FSMs).

The specification in IDL (the language) identifies two subsections of a design: the *logic box*, and the *external world*. The executable portion of the code describes the logic box and its effect on the external world. The non-executable portion of the code describes the structure of the external world. Executable statements include several types of actions that occur in a given state: state transitions; register, bus, and memory transfers; and black box actions. The effect of these actions is carried out by the simulators in order to provide a complete simulation environment. The declarative statements indicate the sizes of registers, busses, memories, and other external objects; organize inputs, outputs, and feedbacks into more easily usable groups; indicate connections

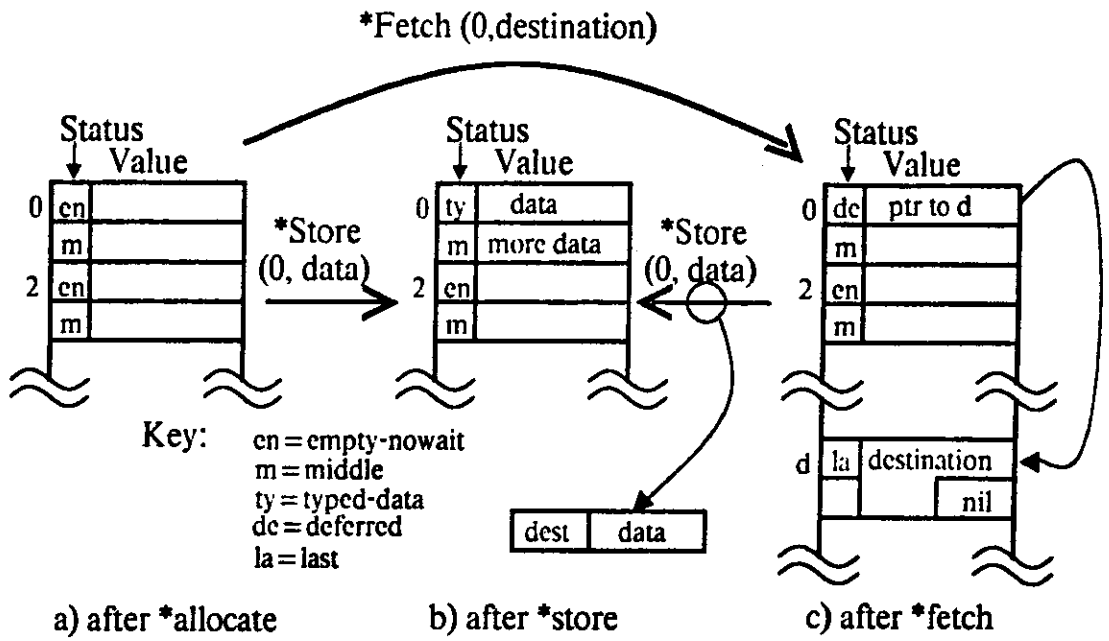


Figure 2-2: Dynamic Behavior of Memory

between the controller and external objects; and allow the user to specify strings for textual substitution.

The compiler makes certain syntax and consistency checks as it translates code into an intermediate form. This intermediate form, called regularized IDL, can be interpreted by the *High Level Simulator*. Regularized IDL can be assembled into two-level logic. This canonical sum of products form can be manipulated in a number of ways (including minimization *a la* MINI [7]), and can be simulated by the *Fast Simulator*. The sum of products result comes in the form of PLA (Programmable Logic Array) personality and can be translated to other forms for conversion into random logic or for various manipulations. Synthesis of a PLA from IDL output is trivial, and paths to other embodiments of the logic can take advantage of the minimized two level form.

3.2. The IDL Language

An IDL designer specifies both the details of the logic being designed and its interaction with other objects (registers, busses, etc.). The first part of an IDL program consists of declarations which indicate the structure of various objects. The following IDL fragment demonstrates two declarative IDL statements.

```
DIM MAR 16/ MDR 32          (1)
FIX CHAIN ← CNTL[4]         (2)
```

(1) declares registers MAR and MDR to be 16 and 32 bits respectively. (2) indicates that the input CHAIN is obtained from the value last latched into the fourth bit of register CNTL.

IDL GROUPs, which are similar to **records** in high level programming languages, allow the designer to refer to several objects at once. IDL STRINGs allow the designer to create mnemonics for less easily understood constructs. The mnemonic may represent a complex condition of the inputs, a value to compare against inputs, or any other textual entity the designer wishes to abstract. STRINGs are resolved by textual substitution before the rest of the program is processed.

The main part of an IDL program is the *executable code*, composed of a series of executable statements. An executable statement has the following syntax:

label: IF <condition> THEN <actions> [ELSE <actions>]

A label can be thought of as a precondition. <condition> may be any boolean function of the inputs. <actions> involve changing state and controlling the external world, and thus include setting outputs, feedbacks, and control lines. The following fragment transfers the contents of REG1 to REG2 (sets a control line to make a transfer) if IN1 is true and if IN2=IN3.

IF (IN1 **and** (IN2=IN3)) THEN REG1 ← REG2

IDL also provides a mechanism (called dot notation) for building decision graphs. A regular label is the root of a decision graph, and dotted labels are the descendants. Moving from one node to another (activating a label) can be accomplished with a goto (→root.child).

The novice IDL user should be careful in dealing with the left arrow or assignment symbol. In IDL, left arrow is used for many purposes.

Declarations:

GROUP declarations:	GRP foobar ← foo, bar
external connections to the PLA:	FIX input ← one-bit-register
string declarations:	STRING boom ← x=1 AND y=0

Executable Code:

register transfers:	reg1 ← reg2
setting outputs:	control ← 1

The IDL compiler distinguishes between the above cases based on the context, but none the less the left arrow is heavily overloaded.

4. ISC Implementation

In designing the ISC, several stages of the design were developed in succession. An informal paper design describing the function and structure of the ISC was the first step. Then a formal high level design in the form of non-executable Pascal-like code was drawn up.† Detailed descriptions of the algorithms were specified, and all of the data structures were then presented. The final stage of the design was the IDL code. The IDL description can be implemented in hardware, and both algorithmic and bookkeeping details are present. Since this code is executable, it can be simulated and verified.

---

†This pseudo Pascal code provided the basis for part of the actual Pascal code that was written by other people in the group to simulate the whole dataflow machine at a functional level on a conventional computer.

The ISC design required about 1500 lines of IDL code. IDL compiled and assembled the ISC to PLA personality (two level logic), and the following statistics were gathered:

Controller:
- Product terms: 887
- Inputs (no feedbacks): 151
- Outputs (no feedbacks): 316
- Feedbacks: 91

Registers: 718 bits
Comparators: 40 bits

The logic derived in this exercise is too big to implement. Also, all of the logic has not yet been compressed to two level form.† The size of the design is partially a result of our approach to the problem; we were more concerned with developing the ISC design and studying IDL, and less concerned with the size of the PLA generated.

4.1. Useful Features of IDL

Several IDL constructs were helpful in the ISC design. In IDL, several cooperating processes can be designed to run in parallel with varying degrees of communication. There are basically two kinds of parallelism that we want to deal with when designing hardware, low level and high level parallelism. Low level parallelism involves actions occurring simultaneously at the lowest level (*e.g.*, "Load register 1 and register 2 in parallel"). By high level parallelism we mean communication between concurrent processes which may be considered sequential as a first approximation. Low level parallelism is handled naturally by IDL statements which allow for the expression of parallel actions. Furthermore, primitives to handle high level parallelism can be easily constructed using available IDL mechanisms. The communication required may come from shared feedbacks, testing and setting the state of another process, or through external connections. The ISC design takes advantage of these facilities to implement three concurrent processes: the token receiving, the instruction execution, and the token dispatch sections. When a **fetch* is taking place there is no reason that the next token cannot be received and the last token sent off, all in parallel. Shared feedbacks were used to synchronize these sections. We were very successful in expressing the high level parallelism of the ISC in IDL. Essentially serial processes were written (more or less) independently and later merged to get the total design. In contrast, the Pascal code does not have any sense of time (other than sequences). As a direct result of this problem, there was no way to express the idea of a hard reset in Pascal.

IDL Truth tables were used to concisely describe several boolean functions that would otherwise have unnecessarily cluttered the code. IDL also allows truth tables for setting outputs in the following manner:

A ← *Truth-Table-Name <input parameters>

Where the truth table may be a multiple output boolean function. The effect of this action is to evaluate the boolean function specified by the truth table and assign that value(s) to A. In the ISC design, the input field carrying the size of the datum (the second half of the type field) was tested in

---

†IDL effectively allows more than two levels of logic by allowing feedback paths with zero delay.

many ways. For example, the precise length of the data has a great bearing on how many words must be referenced while processing a request; truth tables allowed the design to hide the complex boolean expressions from the main source making the code more readable. IDL strings also were a useful tool in the ISC code. By using strings exclusively in output assignment and input testing, we were able to control the ways in which variables were manipulated. This technique is analogous to the renaming of constants in a program. As a result of this organization, a change in the ISC address space from 16 to 24 bits had only a small effect on the IDL design. In a similar fashion, IDL groups allowed data to be reorganized from the bit level into more usable objects. IDL strings, however, are semantically unconstrained entities and allow for the possibility of drastic misuse. Strings can be improved dramatically by using ideas from modern programming theory.

The most useful capabilities of the IDL System are its ability to generate and manipulate two level logic and its ability to simulate and verify the logic designed. The simulation comes in three flavors, allowing the logic designer to verify the design at different levels of abstraction. For several reasons, the IDL code for the ISC was not simulated. The functionality of the design was being verified by Pascal programs derived from the High Level Code. Also, the magnitude of the ISC design severely stressed the IDL system. Simulation using the implementation of IDL as it existed at that time, while possible, would have been difficult. IDL did not, however, have any fundamental problems with the ISC design. If the design were "real", we would have simulated the ISC, perhaps breaking the design into several pieces.

4.2. IDL as a High Level Language

Designing in IDL is a quantum leap above designing at the gate level. Writing IDL is in some ways similar to writing microcode with the aid of a powerful micro-assembler. Yet we were unable to design the ISC directly in IDL without first specifying it in some detail in the Pascal-like pseudo code. The fact that the high level pseudo code was written at all indicates that IDL has not gone far enough in raising the level of abstraction for the hardware designer. The ways in which the IDL specification differs from the Pascal specification indicate some of the difficulties that designers of hardware design languages will face in raising the level of abstraction of a design language. We will explain this point with a few examples.

The IDL code tended to be non-reentrant. When copying a block of code from one part of an IDL program to another, extreme caution had to be used in changing all references to states (labels). Handling errors provides a good illustration of this difficulty. Often a test for a particular type of error had to be made in several places, and in each instance similar actions were required. The only way to express this idea in IDL was by copying code. Modern languages, like Pascal, provide a modern way to handle actions that occur in more than one place. We do not think that this inadequacy of IDL is intrinsic to hardware design languages.

Pascal data structures can not safely be used to directly model the hardware, that is, without losing the advantages of having the data structure in the first place. When a **variant-record** is used in Pascal, there is implied storage (to discriminate the cases). The very bits which are beyond our control in Pascal are the most critical ones. Those are the bits by which decisions are to be made. We must have close control over them or know that the compiler will do a particularly good job in dealing with them. Many important details (from the hardware point of view) were glossed over in Pascal. One of these issues involve passing tokens to the system, and transferring values within it. When we say x:=y in software, no one asks how the value travels. The details of value transferal

cannot be ignored in hardware design.

5. Language Design Issues

The dual exercise of writing the ISC in IDL and Pascal has convinced us of two facts: (1) behavioral design languages such as IDL are essential for hardware design, and (2) sequential languages such as Pascal are inadequate for expressing parallelism, which is present in almost all hardware designs. However, IDL (and we suspect all other hardware design languages in use) lacks many features of modern programming languages, features which are essential to manage the complexity of any large design. IDL was designed more than 10 years ago when many concepts of modern programming languages were not well understood. It has been successful in providing a design tool which allows synthesis of good hardware. Several of the ideas from modern programming languages, especially in the area of data structures, can be incorporated into IDL fruitfully and without much difficulty. However, we discuss two issues related to the quality of hardware generated from a behavioral language that pose some difficulty for all hardware design languages.

5.1. The Enumeration Assignment Problem

The problem of state assignment (discussed extensively in the literature, see for example [10], [12], [11]) is well known and difficult. A similar problem arose in our study of the ISC. In the design of the data-section, eight distinct formats arose for the data word; these formats are specified by the value of the status field: *not-allocated*, *empty-nowait*, *typed-data*, *untyped-data*, *middle*, *invisible*, *type-deferred*, and *notype-deferred*. The actual assignment of binary values to the various enumerations of the status bits have no significance for the functionality of I-store operations. The assignment will, however, have a significant impact on the logic required to operate on these bits.

This Enumeration Assignment Problem is common to the designer of digital systems. The designer must collect all decisions that will be asked of these bits and make an assignment that will optimally implement the decisions. In practice, it is not particularly difficult to make some reasonably efficient assignment. In a design language such as IDL, however, we would like to defer the assignment as much as possible, and allow the assignment to change (automatically perhaps) as we modify the decisions made.

In the design, we wish to operate conceptually in a multiple valued logic. All decisions represent two valued predicates in the larger logic. When the time comes to implement our design in (boolean) hardware, we wish to use the list of predicates to make an assignment that yields the simplest realization of these predicates. The problem is slightly more complex than presented so far. Different variables may interact in such a way that the best enumeration assignment cannot be found independently.

5.2. The Condition Don't Care Assignment Problem

When certain combinations of inputs arise, we may not care which branch of a decision to follow. There is no way of indicating in IDL that a don't care should be associated with a decision rather than an output. Even if we express this fact using some construct, the minimization problem generated by this kind of a don't care for conditions is different from the problem of minimizing

several multi-variable functions (with *output* don't cares).

Consider the following Karnaugh Map (which should be considered a boolean function on two variables) and the code fragment that uses it.

(decision)
Karnaugh Map 1

b \ a	0	1
0	0	?
1	1	1

```
Code Fragment 1

if (Map 1) (a b)
  then x ← 1 / y ← 0
  else x ← 0 / y ← 1
```

Figure 5-1: Karnaugh Map 1

The don't care at the ab=10 position means that both the **then** and the **else** branch provide an acceptable assignment of values to x and y. If we expand the statement into maps for x and y, propagating the don't care, the following Karnaugh maps result:

(action)
Karnaugh Map 2

b \ a	0	1
0	0	?
1	1	1

(action)
Karnaugh Map 3

b \ a	0	1
0	1	?
1	0	0

Figure 5-2: Karnaugh Maps 2 and 3

The don't cares at the ab=10 positions have been decoupled incorrectly in the following sense. There are four ways of assigning the two don't cares, but only two of those assignments preserve the semantics of the decision structure. If the don't cares at ab=10 on Map 1 and Map 2 are both assigned the same value (either 1 or 0), we encounter a peculiar situation. When ab=10, the values of outputs x and y will be equal to each other. These values are inconsistent with both the **then** and **else** branches of code fragment 1 (figure 5-1).

If we want to move the *condition* don't cares to the output side (of the **if-then-else**), we must also carry along some constraints on their assignments. In the example above we must assert that the assignments of the don't cares for x and y must be 01 or 10, hence of opposite values.

We can compress logic to two levels and preserve the semantics of our decision structures as follows. Uniquely label all condition don't cares. When we multiply out the multiple level logic, the don't cares are linked by their labels (some will be in inverted form). Whenever we assign a don't care, all instances of the don't care must be assigned the same value. This is by no means a minimization algorithm. This problem is clearly harder than the "classical minimization" problem.

This problem would never have arisen if we could not express don't cares as a part of conditions. In fact, code fragment 1 (see Figure 5-1) cannot be expressed using only output don't cares. Another syntax for expressing condition don't cares is the **if-either-or** statement:

```
if <condition> then
   either   <actions>
   or       <actions>
```

Even though we don't know how to solve the condition don't care assignment problem precisely, we can apply the heuristic of locally minimizing the boolean map associated with a decision. We might then consider the sizes of the **then** and **else** branches. When a series of decisions are compressed into two level logic, it becomes more important that each decision node is as simple as possible since the size of each term has a multiplicative effect (*i.e.* the terms associated with the *on set* (true condition) have to be **and**ed with the **then**-*subtree*, and *off set* (false condition) terms have to be **and**ed with the **else**-*subtree*. A subtree itself may contain a decision to be resolved. . . .). Hence a don't care involving a *decision* in a decision tree provides the possibility of immense savings. Consider the following Karnaugh maps.

(decision)
Karnaugh Map 4

C \ AB	00	01	11	10
0	0	0	?	1
1	?	0	1	1

assign 00 1 = 0
and 11 0 = 1
so that ON SET = A
OFF SET = $\bar{A}$

(decision)
Karnaugh Map 5

FG \ DE	00	01	11	10
00	0	0	?	?
01	0	?	1	?
11	?	1	1	?
10	?	?	?	?

Figure 5-3: Karnaugh Maps 4 and 5

In the case of Karnaugh map 4, the best assignment seems to be assigning the don't cares in such a way that the *on set* = A, and the *off set* = $\bar{A}$.

The most promising assignment for Karnaugh Map 5 is assigning the 0101 spot to one boolean value, and assigning the rest of the spots to the other value. If 0101 is assigned 0, then the *off set* requires the single term $F \cdot \bar{D}$ and the *on set* requires the term D and the term F. If 0101 is assigned 1, then the *on set* requires the single term $E \cdot \bar{G}$ and the *off set* requires the term $\bar{E}$ and the term G. Depending on the complexity of the **then** and **else** branches, one of the two choices can be made. If the **then** branch is "heavier," we assign the 0101 spot to 1. If the **else** branch is "heavier," we assign the 0101 spot to 0. In any case, assigning all zeros or all ones is probably non-optimal.

6. Conclusions

The design of increasingly complex digital systems requires behavioral design tools such as IDL. These design tools must be able to handle extremely large designs in an elegant manner. It has become clear in performing the dual exercise of expressing the controller for a novel storage

structure in IDL and Pascal that *Pascal-like languages* are not adequate to deal with the subtleties of a complex digital design. IDL has several shortcomings from the point of view of modern programming languages, and most of these weaknesses can be easily fixed. However, there are other shortcomings in IDL when it comes to generating good hardware. We think that these weaknesses are present in all the current hardware design languages and require research for proper solutions.

The novel storage structure described in this paper is of interest in its own right, and a reader not interested in hardware design languages may still find Sections 1 and 2 useful. We think that I-structure storage or similar storage structures will form an integral part of future multiprocessor systems.

Acknowledgments

This work was done during 1982 when Steve Heller was working in Dr. Harold Fleisher's group at IBM Poughkeepsie. We gratefully acknowledge Dr. Fleisher's support and encouragement, and Dr. Leon Maissel's valuable insights into hardware design languages. However, this work could not have been done without Mr. Ray Phoenix, who besides being an IDL expert, knows its implementation best.

References

[1] Arvind and Members of the Functional Languages and Architecture Group.
The Tagged Token Dataflow Architecture.
Technical Report TR (unpublished), Laboratory for Computer Science, MIT, Cambridge, Mass., August, 1982.

[2] Arvind, and Iannucci, R. A.
Instruction Set Definition for a Tagged Token Data Flow Architecture.
Technical Report TM-212, Laboratory for Computer Science, MIT, Cambridge, Mass., December, 1981.
updated May 1982

[3] Arvind, and Iannucci, R. A.
Two Fundamental Issues in Multiprocessing: The Dataflow Solution.
Technical Report Computation Structures Group Memo 226-2, Laboratory for Computer Science, MIT, Cambridge, Mass., July, 1983.

[4] Arvind, and R. A. Iannucci.
A Critique of Multiprocessing von Neumann Style.
In *Proc. of the 10th International Symposium on Computer Architecture.* June, 1983.

[5] Arvind, and R. E. Thomas.
I-Structures: An Efficient Data Type for Functional Languages.
Technical Report TM-178, Laboratory for Computer Science, MIT, Cambridge, Mass., October, 1981.

[6] Heller S. K.
An I-Structure Memory Controller (ISMC).
Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., May, 1983.

[7] Hong, S. J., Cain, R. G., and Ostapko, D. L.
Mini: A Heuristic Approach for Logic Minimization.
IBM Journal of Research and Development 18:443-458, 1974.

[8] Maissel, L. I. and Ostapko, D. L.
Interactive Design Language: A Unified Approach to Hardware Simulation, Synthesis, and Documentation.
In *Proceedings of the ACM IEEE 19th Design Automation Conference*, pages 193-201. June, 1982.

[9] Maissel, L. I. and Phoenix, R. L.
Interactive Design Language.
In *IEEE ICCD.* 1983.

[10] Marcus, Mitchell P.
Switching Circuits for Engineers.
Prentice-Hall, 1975.

[11] Miller, Raymond E.
Switching Theory, Volume II: Sequential Circuits and Machines.
John-Wiley and Sons, Inc., 1965.

[12] Wood, Jr., Paul E.
Switching Theory.
Lincoln Laboratory Publications, 1968.