

DIS: An Architecture For Fast LISP Execution

by

William S. Yerazunis

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

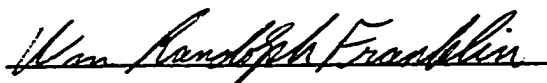
in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Systems Engineering

Approved by the Examining Committee:



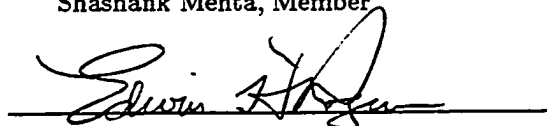
W. Randolph Franklin, Thesis Advisor



Frank DiCesare, Member



Shashank Mehta, Member



Edwin H. Rogers, Member

Rensselaer Polytechnic Institute
Troy, New York
May 1987

© Copyright 1987
by
William S. Yerazunis
All Rights Reserved

Contents

List of Figures	viii
Acknowledgement	ix
Abstract	x
 1 Introduction :	 1
1.1. Why LISP? :	1
1.2. Why special-purpose hardware? :	1
1.3. Why AI compilers and optimizers? :	2
1.4. Proposed New Architecture :	2
 2 Background and Machine Survey :	 3
2.1. Critique of Conventional Architectures :	3
2.1.1. Shortcomings of Conventional Architectures :	3
2.1.2. Shortcomings of Vector Architectures :	4
2.1.3. Lack of Multiprocessor Support :	5
2.1.4. Defects with Current LISP Designs :	6
2.2. Definition of Terms :	6
2.3 Machine Survey :	8
2.4. SIMD Processors :	8
2.4.1. The Bombe :	8
2.4.2. ILLIAC IV :	9
2.4.3. MPP :	9
2.4.4. STARAN :	10
2.4.5. Systolic Arrays :	10
2.4.6. Vector Processors :	11
2.4.7. Connection Machines :	11
2.5. MIMD machines :	12
2.5.1. Cm* :	12
2.5.2. EMPRESS :	13
2.5.3. ZMOB :	13
2.5.4. Ultracomputer :	14
2.5. LISP machines :	15
2.5.1. Xerox PARC :	15
2.5.2. M.I.T. CADR derivatives :	16
2.6. Multiprocessor Lisp machines :	16
2.6.1. Evlis :	16
2.6.2. MEF :	17

2.6.3. AMPS :	17
2.6.4. AHR :	17
2.7. Other Architectures :	18
2.7.1. The Warren Prolog Machine :	18
2.7.2. VLIW Architectures :	18
2.8. Summary of Current Architectures :	19
 3 Applicative Programming and Parallelism :	 22
3.1. Pure applicative programming :	22
3.2. Applicative Programming Variants :	23
3.3. Applicative Result Passing :	23
3.4. Single-Assignment Programming :	24
3.5. Dataflow Programming :	24
3.6. Loosely Applicative Languages :	25
3.6.1. Applicative LISP :	25
3.6.2. Non-Applicative LISP Functions :	25
3.7. Computability of Applicativeness :	26
3.8. Costs of Applicative Programming :	27
 4 The LISP Language :	 28
4.1. Parenthesis :	28
4.1.1 Prefix Notation :	28
4.1.2. Argument Passing :	29
4.1.3. Program Representation :	29
4.2. Variable Scoping :	29
4.3. Functions as Arguments :	31
4.4 An Abstract LISP Engine Design :	32
4.4.1. Data Needs of LISP :	32
4.4.2. Needs for Parallelism :	32
4.4.3. The Abstract LISP Supercomputer - DIS :	33
 5 System Design Constraints :	 34
5.1. Uniprocessor/Multiprocessor :	34
5.2. Maximum Processors :	34
5.3. Language Orientation :	34
5.4. Package Pinout :	34
5.5. Gate Delay Speed :	35
5.6. Network Interconnect :	35

6 Design Methodology :	36
6.1. Ranking Methods :	36
6.2. Orthogonality versus Preferentiality :	36
6.3. Iterative Design :	37
6.4. Functionality Provided :	37
6.5. Final Comments on the Design Process :	38
7 The DIS Processor Design :	39
7.1. Basic Modules :	40
7.1.1. Memory :	42
7.1.2. Stacks :	43
7.1.3. DIS Compiler Stack Usage Model :	45
7.1.3.1. Value Stack Usage :	45
7.1.3.2. Pending Stack Usage :	45
7.1.3.3. Return Stack Usage :	47
7.1.3.4. Binding Stack Usage :	48
7.1.4. ALU :	48
7.1.5. Sequencer :	50
7.1.5.1 Next Address Processing :	50
7.1.5.2. Command Word Distribution. :	51
7.1.6. Network Interface :	52
7.1.7. Booting, Debugging, and The Host :	53
7.2. Intraprocessor Interconnection and Parallelism :	54
7.2.1 Construction of the Crossbar :	55
7.2.1.1. Option 1: Brute Force Crossbar :	55
7.2.1.2. Option 2: Microwave or Fiber Optic :	56
7.2.1.3. Option 3: Restricted Configuration Crossbar :	57
7.3. Conditional and Call Instructions :	58
7.4. Garbage Collection in DIS :	59
7.4.1. Estimate of Time to Garbage Collect :	59
7.5. Concluding Design Remarks :	61
8 The Compiler, Optimizer, and Simulator :	62
8.1. System Definition Macros :	63
8.2. Compiler Internals :	63
8.2.1. Recursive Descent Modules :	64
8.2.2. Symbol Tables :	65

8.2.3. Code Generator Stubs :	65
8.3. Optimizer Internals :	66
8.3.1. Basic Block Partitioning :	67
8.3.2. Clash Determination :	68
8.3.3. Rule-Based Merger :	69
8.4. Linking Assembler/Loader :	70
8.5. Simulator Internals :	70
8.5.1. Order of Execution :	71
8.5.2. Functional Unit Simulations :	71
8.5.3. Simulation Support Functions :	72
8.6. Testing the DIS Software :	72
9 Examples of DIS Machine Code :	74
9.1. Addition of two literal values :	75
9.2. Addition of two variables :	78
9.3. Nested Routine Calls :	80
9.4. Recursive Functions :	83
9.5 Parallel code execution :	88
10 Tests of the DIS architecture :	91
10.1 Benchmarked Machine Implementations :	91
10.1.1. DIS Machine Simulator Assumptions :	92
10.2. Testing Environment :	92
10.2.1. Garbage Collection Time Not Included :	93
10.2.2. Paging and I/O Time Minimized but Included :	93
10.2.3. Runtime Data Type-Checking Disabled :	93
10.2.4. Enabling of Compiler Optimizations :	94
10.3. The Benchmark Suite :	94
10.4. Benchmark Results :	96
10.4.1. Method of Obtaining Timings :	97
10.4.2. Measured Results :	97
11 Conclusions and Further Research :	100
11.1. Specific Goals Achieved :	100
11.1.1. What Features of DIS Cause the Speedup? :	101
11.2. Hardware Implementation Issues :	101
11.2.1. Crossbar :	102
11.2.2. Memory Arrays :	102

11.2.3. Memory Indexing :	102
11.2.4. Cache :	102
11.3. Possible Defects in This Research :	103
11.3.1. Benchmark set too small :	103
11.3.2. Compiler "Caters" to the Benchmarks :	103
11.3.3. DIS Does Not Do Interrupts :	103
11.4. Hindsight Comments :	104
11.5. Directions for Further Research :	104
11.5.1. How Well Does DIS Utilize Multiple CPU's? :	104
11.5.2. Does DIS Work Well with other Languages? :	104
11.5.3. Will a Wider DIS Machine go faster? :	105
11.5.4. Can DIS run non-procedural languages? :	105
References :	106
Index :	113
Appendix A (DIS File Listings) :	122
Appendix B (DIS Instruction Summary) :	198

List of Figures

2.18a Venn Description of Computer Architecture	20
7.0a A Single DIS Processor	39
7.1.1a A DIS Memory Unit	42
7.1.2a A DIS Stack Unit	44
7.1.3.2a Data Flow During Function Call	46
7.1.4a The DIS ALU Unit	49
7.1.5.1a The DIS Sequence Address Generator	51
7.2.1.3a Restricted Crossbar	57
7.3a The DIS User-Visible Pipeline	58
8.0a DIS Simulation Overview	62

Acknowledgements

The author wishes to express his thanks to the following persons and groups, who, each in their own way, helped shape the research described here:

Stephen Yerazunis, Mary Yerazunis, W. Randolph Franklin, Joseph Mundy, John Tribble, Eric Luce, Randall Shane, Paul Charlton, Frank Lynch, Digital Equipment Corporation, John Barnwell, Frank DiCesare, Ed Rogers, S. Mehta, Charles Close, The RPI Electrical, Computer, and Systems Engineering Department, The RPI Computer Science Department, Ken Connor, John Barnwell, Bill Barabash, Chris Connolly, Josh Fisher, John Ellis, Phil Lewis, Danny Hillis, Tom Knight, Walter VanRoggen, Paul Anagnostopoulos, A. Yavuz Oruc, Ken Walters, Richard Welty, The General Electric Company, Eirikur Hallgrimmsson, Carol Jens, Richard Glackemeyer.

This document was typeset with \TeX (courtesy Donald E. Knuth), and the figures were drawn with SIGHT, on a DEC AI VAXstation (courtesy Digital Equipment Corporation).

DIS: An Architecture for Fast Lisp Execution

Abstract

DIS is an architecture for very fast execution of LISP and other artificial intelligence languages. The DIS architecture uses a number of functional units controlled by a wide (256 bit) instruction. A simulator, compiler, and optimizer have been constructed for the DIS architecture. A simulated 100 nanosecond cycle time single-processor DIS machine appears to run LISP on the order of twice as fast as a CRAY-1, and on the order of ten to fifteen times faster than other LISP-directed architectures.

William S. Yerazunis

Chapter 1

Introduction

In this thesis we will pursue the question of how to execute LISP quickly. We will attack this problem with a number of tools, such as special-purpose hardware, high-bandwidth memories and interconnects, and artificial intelligence techniques in the compilers and optimizers. Our target is to design a hardware and software system that uses non-exotic technology but runs LISP faster than the most powerful supercomputers existing today. The simulated DIS machine runs a test set of LISP programs about twice as fast as the Cray-1, and ten to fifteen times faster than other LISP-directed architectures.

1.1. Why LISP?

LISP was chosen as the target language for this thesis for a number of reasons. LISP is one of the few popular languages which is applicative in nature, and we find that we can use this applicative nature to achieve parallelism.

LISP is a language well suited for AI-based manipulation of program structures, because all LISP functions are also LISP data structures. LISP programs can manipulate other LISP programs easily and effectively. We will find that this automatic manipulation of program structure by compilers and optimizers is necessary to achieve parallelism over a wide variety of programs.

1.2. Why special-purpose hardware?

We chose to design special-purpose hardware to take advantage of the rapid decrease in memory prices. In this design, we found cases that implied a single additional memory could increase our speed by a factor of two (such as separate memories for instructions and stack space). Other cases (such as separating data memory from symbol table memory, separating stack data from stack control information, etc.) all showed great promise for increased speed.

No commercial design met our needs for this high-bandwidth multiple-ported memory. No commercial design had the internal data path flexibility to allow the DIS compiler enough degrees of freedom to use the bandwidth, even if the bandwidth was available. Commercial designs had been running LISP compilers for twenty years, and we felt it was unlikely that we could do any better with a conventional datapath and single memory.

Our feelings were vindicated by the results of testing the DIS compiler on actual LISP code; the compiler is generally able to keep at least four different operations happening at the same time. Often the compiler does better than this. Having hardware with less internal flexibility would make the intelligent compiler useless.

1.3. Why AI compilers and optimizers?

We chose to use the artificial intelligence techniques of forward and backward tracking [Tjaden 1970], and both conflict-based and rule-based operations, because conventional compiler technology has been notably unsuccessful in automatically detecting parallelism in programs. Most commercial parallelizing compilers require the user to insert special vectorization statements [Hoffman], which severely limits the usefulness of those compilers. We wanted automatic detection and utilization of as much parallelism as the compiler and optimizer could find. We feel that we were moderately successful in detecting and utilizing the available parallelism.

1.4. Proposed New Architecture

This thesis describes in detail the DIS machine, both the hardware architecture and the system software.

The DIS architecture provides hardware support to minimize the number of instructions needed to execute typical LISP functions. The DIS architecture supports parallelism at multiple levels. Multiple functional units operate concurrently within a single processor, and provision is made for multiple processors to be used.

Chapter 2

Background and Machine Survey

This section discusses current designs for commercially available high-performance computers. Machines which internally overlap instructions in a non-program-visible way will be considered equivalent to purely serial general purpose machines. Machines which have program-visible overlapped or parallel instructions such as vector instructions will be considered separately.

Under the program-visibility criterion, all current supercomputers are vector machines. By appropriate use of the vector instructions a great increase in speed is obtained for certain numerical array oriented problems. Speedups on the order of fifty to one hundred times are experienced for certain problems.

2.1. Critique of Conventional Architectures

Unfortunately, call-intensive languages such as LISP cannot be efficiently executed on vector supercomputers. This is because the vectorization hardware does not include subroutine call as a vectorizable operation. Several language oriented machines are commercially available for execution of LISP, but the price/performance ratio is poor, a single-user machine such as a Symbolics 3670 costing approximately \$100,000.

2.1.1. Shortcomings of Conventional Architectures

General-purpose computers often have a number of high-speed registers located within the CPU. These registers fall into two classes: dedicated registers, such as the program counter and stack pointer, and general purpose registers, which are used by the program for arbitrary purposes. Each register's contents in the CPU must be saved in memory when a subroutine is called, and restored properly when the subroutine returns (unless the subroutine returns the result in a register, then care must be taken NOT to overwrite the result).

Because modern designs often have large numbers of available registers in the CPU, it is not unusual to have subroutine call be one of the slowest instructions in the processor. Many memory cycles must be used to save the registers to memory. For example, a VAX has over half a kilobit of user-visible register storage, all of which must be saved under the "caller-saves" convention [Digital 1979]. Even if memory interleaving is "caught" correctly, eight doubleword transfers must still be run, before the first instruction of the subroutine may be executed. It may be concluded that large CPU-resident, program-visible register files are an impediment to execution of structured programs.

Conventional architectures are often designed with non-program-visible parallelism, such as overlapped instruction fetches, instruction decoding, operand fetches, and instruction execution. These overlapped operations provide some speedup at the

cost of extra hardware needed to interlock the user program against faulty execution due to timing mesh. For example:

```
Load Reg1, Memory(123)
Inc Reg1
Store Memory(123), Reg1
Load Stackpointer, Memory(123)
```

has results which are clearly dependent on whether or not the Store is completed before the operands are fetched for the Load Stackpointer. The machine designer must decide whether to provide interlock hardware or to require the software to obey more restrictive rules concerning instruction sequencing. Nearly all designers choose the extra hardware option [Norrie 1984], [Tornig 1984]. Doran gives an excellent method for determination of optimal scheduling of various stages of an instruction by the use of Group Theory [Doran 1979].

2.1.2. Shortcomings of Vector Architectures

To provide better support to large numerical analysis problems such as weather simulation, finite element modeling, and image processing, the "vector processor" has been pursued. The concept is that the elementary data structures the machine deals with at the single instruction level are extended to include the one-dimensional array. Instructions which use one-dimensional arrays as arguments are included in the instruction set. The use of these vector instructions is very common in the supercomputer business; so much so that the terms "supercomputer" and "vector computer" are often used interchangeably in the literature.

The Cray Research Cray-1, Cray X-MP, Cray-2 and Control Data Corp. Star-100 and Cyber 20x series machines are of the vector type. The Cray-1 has sixteen vector registers, each vector register holds sixty-four numbers, each number is sixty-four bits long. This large amount of data is not saved when a subroutine is called; it is the responsibility of the using subroutine to preserve the vector registers. The CDC Star and Cyber 205's use a strictly memory-to-memory approach when dealing with vectors. This approach drastically decreases the responsibility of the call/return instruction but increases the setup time to execute a vectorized instruction.

Unfortunately, the current supercomputers are super-specialists. Vectorized instructions are generally restricted to the form (for Cray machines)

```
for i = 1 to 64
  C(i) = A(i) + B(i)
next i
```

or CDC vector machines (n and j being 16-bit integers):

```
for i = 1 to n step j
  Z(i) = S * X(i) + Y(i)
next i
```

No current commercial vector computer can execute the following example in vector mode (A and B being integers, X, Y, and Z being vectors of integers):

```
Function MyAdd ( A, B)
MyAdd = A + B
return
end

for i = 1 to 10
  Z(i) = MyAdd (X(i), Y(i))
next i
```

Another deficiency is that the vector computer is nowhere near as "super" in non-vectorized mode as it is in vectorized mode. For example, a Cyber 205 supercomputer, when equipped with the maximum number of floating point units (pipes) can execute about 800 million floating point operations per second (for long vectors). However, the 205 can only execute about 20 million non-vector instructions per second. This ratio of 40:1 is not atypical; the Cray X-MP is capable of about 480 million vector instructions per second vectorized and only about 10 million instructions per second nonvectorized [Hack 1986].

One reason for this discrepancy is that different hardware is used for vector operations than for scalar operations. The repeat-counts and vector pointers are not available for use by any but the vector instructions. Furthermore, most vector instructions are restricted to floating point operations, (in some cases, to floating point operands stored in special "vector registers" of fixed length). No provision exists to allow an "increment-and-repeat" instruction modifier.

2.1.3. Lack of Multiprocessor Support

Neither general-purpose computers nor vector supercomputers are not designed with mass multiprocessing in mind (having tens or hundreds of processors all operating concurrently). A general-purpose computer might supply a read-modify-write synchronization primitive, and many supercomputers have no synchronization primitive at all. In CDC machines, synchronization and process exchange is carried out by peripheral processors halting and restarting the fast main processor. Context change not only must save all program-visible registers, but must also save the process-dependent information such as page table locations, virtual memory state, and system protection and security information, making context change extremely expensive and slow.

Some multiprocessor machines have been commercially successful, but it has been questioned whether any system makes good utilization of additional processors [Fuller 1978]. The literature indicates that it is about average to obtain performance of 1.6 times the uniprocessor performance with a dual processor. A large proportion of this is due to the cost of interlocking the processors so that no task is lost, and no task is executed more than once [Agrawala 1983].

2.1.4. Defects with Current LISP Designs

In order to support call-intensive languages such as LISP, special purpose architectures have been designed. These architectures typically use a large vertical microcode to map a virtual LISP machine onto a relatively simple physical machine [Chu 1981].

LISP processors are commercially available. A typical system costs \$100,000, and each system can serve only one user at a time [Weinreb 1981]. Xerox produces the 1108SIP for \$20,000, but hardware limitations (maximum memory and disk size restrictions) make the 1108 less useful than the other commercially available systems.

A further disappointment is that the special purpose LISP machines are slow. Typical designs use ECL in the processing unit, but due to the vertical microcode the performance barely approaches 1 million machine instructions per second, let alone one million LISP functions per second [Weinreb 1981]. It is not the case that these machines are implemented with slow technologies; rather, these systems are saddled with a large microcode to implement a complex system (LISP) on an architecture which is not much more powerful than a PDP-11. Overheads of two hundred microinstructions to execute a function call are not unheard of [Lampson 1981]. The microcode stores are also complex (and therefore error-prone), being on the order of 6000 words to implement the core set of instructions, which then provide the basis to execute LISP functions.

Finally, current LISP machines do not have multiprocessor capability. When a problem becomes sufficiently complex that it overloads a single processor, the only choice is to go to a faster processor. There is no option to add a second processor to speed things up.

2.2. Definition of Terms

SISD - Single Instruction Single Data: A computer design which specifies that one stream of instructions is applied to one set of data. Most general purpose computers are of this type. Each instruction specifies a particular manipulation of one piece of data, and the instructions are carried out serially (or appear to be carried out serially, from the programmer's point of view)

SIMD - Single Instruction Multiple Data: A computer design in which a single instruction stream performs manipulations on multiple sets of data. Each data manipulation specified in the instruction stream is executed by one of a number of processors, each one containing one data set. All processors perform the same operation at the same time. This is usually insured by having a central facility read the instruction stream and broadcast the current operation to all data processors in the system.

MIMD - Multiple Instruction Multiple Data: A computer design in which a number of different instruction streams are paired with distinct data sets, and the

instruction streams are executed concurrently on multiple processors. MIMD machines have the greatest flexibility (they can emulate both SISD and SIMD machines) but the programming of the multiple instruction streams for coordination of tasks is nontrivial, and in certain architectures, very difficult.

Cache - A cache is a small, fast memory placed between a storage requestor and a storage server. On every request to memory, the cache intercepts the address requested and looks in an internal table to see if that word is stored locally in the cache. If the word is local, the cache returns the desired word to the storage requestor. If the word was not local, the cache commands the main memory to fetch the word. The advantage to cache is that the cache memory, being smaller, can be implemented with a much faster (and more expensive) technology; perhaps as fast as the internal processor data paths. Cache design requires the use of a cache control strategy. This strategy must determine when to load a memory word into the fast cache memory, (thereby taking up one of the relatively expensive cache words) and when to write the word to the memory server. Caches are usually demand-driven: the most recently demanded words are kept in the cache.

Stack - A register-like device which supports not only the operations read and write, but also the operations push and pop. Push is defined as "store the current value in a safe place, and make the new current value the input". Pop is defined as "throw away the current value and make the value from the 'safe place' the new current value". Push and pop are defined to be usable repeatedly; an arbitrary number of pushes followed by the same number of pops should leave the original value on the stack.

Virtualization - The process of making a system appear to be larger, faster or equipped with a new instruction set without actually enlarging or improving the system. For example, virtual memory makes the memory system appear to be very large, without adding more memory devices. Instead, the memory devices already in the system contain only the most often used areas of the memory data, and when a memory location that is not present is requested, a delay is incurred while the memory location is fetched from a slow bulk store (like a disk). The only program-visible effect of this virtualization is that some memory locations may be very slow to access. Likewise, when using a virtualized instruction set, some "instructions" may really cause unimplemented instruction interrupts, which are then simulated with software subroutines.

Von Neumann Architecture - A computer design in which no attempt to differentiate program storage and data storage is made. In fact, von Neumann machines are often touted for their ability to manipulate programs as data and data as programs. Von Neumann machines typically view memory as a large number of cells, each of which may be read and written many times. In the course of program execution, each cell may take many values. There is no concept of "unalterable" storage in the strict von Neumann computer.

Applicative Architecture - A computer design in which no alteration to data storage or program storage is permitted. All calculations are performed by "applying" a function to a piece of data, and the result is not returned in data memory,

but (usually) put on a stack. Later calculations may use the value returned by a previous calculation but may not alter the value. A completely applicative architecture cannot be programmed (because it is forbidden to alter program store) but a mostly-applicative computer has certain advantages over a von Neumann (everything alterable) design, in terms of memory protection and ability to use multiple processors without concern that processor A and processor B may be engaged in a "race" to alter a common memory location. Applicative programs or subroutines are also called "side-effect-free" because they do not alter any element of storage, they only return a value based upon some calculation using those data elements.

2.3 Machine Survey

This section deals with the previous machines which have implemented multiprocessor program execution [Schwartz 1983b], and LISP execution. We will examine these machines both as to how they fall in the SIMD/MIMD taxonomy, and as to special features these machines have in order to support parallelism and LISP execution.

The reader should note that there is a comparative dearth of machines which are both multiprocessor machines and LISP machines.

2.4. SIMD Processors

This section deals with specific single instruction multiple data computers. Particular shortcomings of each design are noted.

2.4.1. The Bombe

An early electromechanical multiprocessor was the Bombe, designed by Rejewski at the start of World War II to crack the German Enigma code by exhaustive search of the key space [Hodges 1983]. Each unit of a Bombe contained six shafts driven by an electric motor. Each shaft drove six modified encoding units. Each encoding unit generated the encoded form of a three-letter code predecessor. When the output of any encoding unit matched the known code predecessor, a relay tripped which turned off the electric motor and applied a brake to the shafts. The correct code key could then be read from the shaft position counters of the triggering encoding unit.

The Bombe was a SIMD fixed-program machine. All of its elements executed the same program (encode and test) at the same time, on different data. There was only one branch allowed, a branch to a global HALT when a solution was found. This general scheme (having a number of processors execute exactly the same program on different data) has been used in computer design ever since.

2.4.2. ILLIAC IV

ILLIAC IV, a "modern" computer in many senses, shared many shortcomings of the Bombe. ILLIAC IV could use up to 64 (256 in the original design) processors, each executing instructions broadcast to the processor array by a central controller. The processor array was designed as a square, with each processor connected to the four nearest neighbors. Under software control, edge processors could be connected to the opposite edge, creating a torus, or the processor array could be electronically switched into a linear array of 64 processors [Hoffman], [Seban 1984]. ILLIAC IV became operational in 1972 and was dismantled in 1985.

Individual ILLIAC processors had no independent control logic except that which allowed it to execute or ignore an instruction based upon the contents of an internal 8 bit flag register. Branches were performed by inhibiting one subset of processors with the flag registers, broadcasting the first branch's instructions, inhibiting that branch's processors, enabling the previously inhibited processors, and broadcasting the second branch's instructions. This control mode decreased the speed of the machine drastically. And like the Bombe, ILLIAC's individual processors could only signal the central control via a global control wire which all processors shared.

Interprocessor communication in ILLIAC was similarly primitive: each processor was connected to only its four nearest neighbors, and there was no provision to communicate over greater reaches without having the instruction broadcast unit perform the data transfer with the entire processor array in a halted state.

Despite these limitations, ILLIAC IV was used for considerable work in the areas of fluid flow and heat transfer modeling, where the short communication distance and relative lack of branching were not great disadvantages.

ILLIAC IV provided useful insights into the programming methodology of processor arrays.

2.4.3. MPP

In order to process larger problems, NASA contracted for the production of a Massively Parallel Processor, or MPP. The MPP (located at the NASA Goddard Space Flight Center in Greenbelt, Maryland, and operational since 1984) contains an array of 16,374 single-bit processors, which can be connected as a square array, a vertical or horizontal loop, a torus, or a twisted torus, all under program control. A spare bank of 4 x 128 processors is provided for redundancy, and can replace any of the 32 4x128 processor columns. As in ILLIAC IV, instructions are broadcast to the entire processor array by a central controller. Like the Bombe and ILLIAC IV, the MPP has a one-bit signal readable by the control processor, which is the global OR of a status bit in each processor in the array [Batcher 1980], [Edelson 1984].

The MPP can execute in excess of 6 billion 8-bit additions or 400 million 32-bit floating-point additions per second, provided the structure of the problem can be mapped onto the available array configurations. Since the processors are all one

bit wide, even the addition of two two-bit numbers must be accomplished by a subroutine.

2.4.4. STARAN

The Goodyear Aerospace STARAN processor array, although a temporal predecessor of MPP, exhibits several useful features not present in MPP. The STARAN array consists of multiple banks of 256 bit-wide processors; a typical STARAN contains 1024 such processors. Only a few state bits are associated with each processor [Batcher 1982]. The first STARAN became operational in 1972. STARAN systems are commonly used for multiple-target radar tracking and other military applications.

The processor array is connected to a very wide memory of 256 bits per processor bank, via a switch called a "flip unit". The flip unit maps a processor address onto a bit-in-memory address, in a way programmed by the control processor. Each STARAN processor can directly access the entire 64K bits within the local STARAN bank.

STARAN processors can pass information via the memory and flip units, or in the ILLIAC style nearest-neighbor routing. The STARAN is connected linearly only, so the only obvious data transfers are to the left and to the right. These bit-passings propagate across STARAN bank boundaries, and they do use the flip units, so that relatively arbitrary interconnections may be achieved.

The STARAN was not designed to perform fast floating-point calculations, rather, it was designed to track multiple radar targets simultaneously. Perhaps a better way to consider a STARAN is not as a large number of small processors, but a block of programmable associative memory. The STARAN design concept has been extended to other machines, including a STARAN-equivalent processor of roughly one cubic foot, for airborne applications [Christ 1984]. This concept of programmable associative memory can also be seen as the basis for the Connection Machine [Hillis 1985].

2.4.5. Systolic Arrays

The above systems all use the same model for computation; many processing elements, all elements doing the same instruction at the same time. It has been pointed out that this is not the only way to do parallel computation; Zakharov [Zakharov 1984] shows that a pipeline of communicating processing elements provides parallelism as well. In a pipeline, each element reads an input from a predecessor processor, performs some computation, and writes an output for successor processors to utilize.

Ordinarily, processors in a pipeline are memoryless. If local storage is included in the pipeline processors, the resulting design is called a systolic array.

H.T. Kung's method for systolic arrays provide a design basis for construction of arbitrary pipelined functions. A one-dimensional systolic design may be constructed of a series of programmable blocks. Each block receives information only from its nearest neighbor on the left, and transmits information only to its nearest neighbor on the right. Each block is independently programmable, and there is a method of synchronization to keep the blocks in step. The result of a computation may be stored in the local memories of the blocks, or it may be emitted from the last block in the chain.

Multiple-dimensioned systolic arrays may be constructed by similar rules of communication. The multidimensional systolic array may be based on other grid patterns than the square, such as hexagonal or body-centered cubic.

2.4.6. Vector Processors

Vector processors such as the Cray-1, Cray X-MP, BSP, and Cyber 205 exist, but these machines are completely inflexible concerning the operation to be performed on the data array. If a processor was not specifically designed to accommodate a particular manipulation, then that manipulation cannot be performed in high speed (vector) mode. For example, the Cray systems do not have a "divide" instruction for vectors. All divides are accomplished by reciprocal approximation and then multiplication. If the result is needed to some accuracy, then several convergent reciprocal approximations are used [Kuck 1982], [Lincoln 1982].

2.4.7. Connection Machines

Thinking Machines, Inc. has built and is offering for sale a SIMD processor, known as a Connection Machine, designed for manipulation of data structures known as xectors (eXtended vECTORS). A pair of xectors may have some one-to-one association between their elements, called a xapping (eXtended mAPPING). Composition of xectors into xappings may be done with an arbitrary function (currently a LISP routine in the Thinking Machines Inc. product), so that the xapping operator can be utilized to sort, sum, cull, or select data objects in a xector. System software also provides the ability to appear to replicate a single data object a large number of times (without actually doing so), and to manipulate xapping type objects as inputs to other xapping operations. Details of connectionist designs are to be found in [Bawden 1984], [Hillis 1985], [Knight 1984], and [Waltx 1987].

The software which provides this capability is mapped onto a 64K processor array with a pair of interconnection systems, with overall control done by a Symbolics 3670. Each processor is only one bit wide, with 1Kbit of local storage. A special controller (itself controlled by the 3670) broadcasts instructions to the processor array. Like ILLIAC IV, each processor can perform the instruction broadcast, or ignore it. Each of the 64K processors has access to a WIRED-OR status line, so that the controller can query the entire processor array.

Interprocessor communications in the connection machine is performed on an n -dimensional ($n = 16$ in the current design) cube. A message to be sent to another

processor is prefixed by the address of that processor. Each communications controller stores and forwards messages. Because of the n-dimensional communication network, the distance (in hops) a message must travel between any two nodes is equal to the Hamming distance between the two node addresses.

The major difficulties in the connectionist design are that the communication network is too slow, and that the current (but hopefully not future) design is SIMD. The vector/mapping approach generalizes the vector/array processing approach to allow any function which does not contain a conditional element as the kernel operator. If the kernel operator contains a conditional, then the Connection machine must serially execute all of the possible paths.

2.5. MIMD machines

This section is deals with various systems of the multiple-instruction multiple-data form. These designs are capable of executing more than one instruction stream simultaneously.

2.5.1. Cm*

Both C.mmp and Cm* are based upon the DEC PDP-11 computer architecture. The C.mmp uses a shared-memory system and PDP-11/03 processors, while Cm* uses a circuit-switched hierarchical bus memory system and PDP-11/23 processors. In other respects, C.mmp and Cm* are nearly identical, and only the more recent (and flexible) Cm* will be discussed [Fuller 1978].

Each of the processors in Cm* is a full-fledged computer, except that only a few processors have I/O devices attached. Memory, although distributed with the processors, appears to all processors as a large, unbroken linear array, shared by all the processors. The only program-visible effect is that memory addresses have varying access times, depending on where the addressed word physically resides, relative to the requesting processor.

Each of the processors in Cm* executes a program which (preferably) resides in the local store of that processor. If a processor accesses a nonlocal word, the processor waits for the hierarchical bus network to acquire the contents of that word. A prefix mechanism is provided in the network hardware to allow each processor to map other processor memories in any way that the programmer desires.

Despite the flexibility of this scheme, the difficulty in programming Cm* is in dividing up the problem into a series of tasks which may execute concurrently. Task allocation is done by the human programmer, as well as the mapping of other processor memories into the address space of each processor.

No synchronization primitives are given in the hardware to allow convenient multiprocessing; the interlocks which cause a spawned process to execute on one and only one processor are done in software. This restriction makes process spawning

an extremely expensive event. Cm* literature varies in the quotations for process spawn overhead; however, no report indicates any reason to believe that more than 40 processors could be used on any problem, before the overhead of processor control became the dominant factor in run time.

2.5.2. EMPRESS

A system similar to Cm* is EMPRESS, at the Federal Technical Institute, in Zurich, Switzerland. EMPRESS is based on the PDP-11 processor like Cm*, but has a control processor and 16 slave processors. EMPRESS was specifically optimized for execution of simulations written in the language Parallel Power-Series Continuous Simulation Program. Since there is a control processor, a bottleneck occurs in synchronization when tasks in independent slave processors wish to exchange information. A significant part of the control processor's time is spent keeping track of which slaves are available, what data is available, and what tasks have not yet been assigned [Buehrer 1982], [Manner 1984].

2.5.3. ZMOB

ZMOB, a multimicroprocessor, uses a circular structure for both information transfer and process control. A complete ZMOB consists of 256 Z-80 microprocessors, arranged as successive stages on a 48-bit wide shift register. Every 100 nanoseconds, the register shifts the value in processor n to processor $n+1$. The shift register has 257 stages; the extra stage is used by the host processor [Kushner 1982].

Each Z-80 in ZMOB is a complete system. There is a 1K ROM which boot-loads the Z-80, and 63K of RAM. Each Z-80 also has a floating-point unit (AMD9511) and an asynchronous serial I/O device. The individual Z-80s run CP/M.

ZMOB, like Cm*, does not have hardware support for spawning of processes. To start a process running in another processor requires either cooperative software running in both processors or the intervention of the control processor (a Vax). Although the shift register provides a general communication network, ZMOB has been promoted for image processing applications, because the cooperating software problem is simpler.

2.5.4. Ultracomputer

Based on work by Schwartz on the idealized model of a paracomputer, NYU is developing a multiprocessor known as the Ultracomputer. The paracomputer model is a model that allows a large number of readers and writers to access a memory word without conflict. Each reader returns a word which was either the value of the word before any of the writers accessed the word, or else one of the words written by a writer task. The paracomputer model specifically prevents "partial writes", where only part of the word has been written by a writer task before a reader task reads the location. Thus, the paracomputer model requires that the final result of some set of simultaneous reads and writes be the same result as some possible serial execution of the same reads and writes [Gottlieb 1983], [Schwartz 1983a], [Stone 1984].

In order to implement this model, the Ultracomputer has a large memory (divided into modules) attached to a number of processors via a switching network. The network is of the Banyan type, but with the addition that the nodes of the switch are locally intelligent. Because of the paracomputer model of serialization, the intelligent switch node can discard certain operations. For example, if a switch node notes two writes to the same location, one of the writes may be thrown away. This can be done because there exists at least one serialization in which the first write is immediately followed by the second write. Since there is no intervening read, no possible reader task could have seen the first write value in the location. The first write is therefore superfluous and may be discarded.

In like manner, the intelligent switch nodes can combine read operations so that switch nodes deeper in the Banyan tree see only one read, rather than two. A read and a write operation to the same location can return the argument value of the write, because there is at least one serialization in which the read immediately follows the write and therefore the read will get the value of the write.

The intelligent switch nodes also are necessary to the "fetch and add" instruction provided for task dispatch in parallel. Fetch and add of n and q is defined to the user as "fetch location n , return me the value there, and store the fetched value $+ q$ in that location, without allowing any other task (readers or writers) access to the location for the duration of the fetch, addition, and store". This instruction provides a fast process dispatch mechanism with a minimal amount of software.

One can observe that fetch-and-add is combinable at a smart network node. For example, if processor 14 fetch-and-adds location 2 with $q=1$ and processor 23 fetch-and-adds location 2 with $q=1$, the network node can pass along a message to fetch-and-add location 2 with $q=2$. The result is returned to the network node, which then returns the value of the fetch-and-add to processor 14, and the fetch-and-add $+ 1$ to processor 23. The memory module containing location 2 has only serviced one read request, not two, and each of the processors see a result which satisfies the Schwartz paracomputer serialization model.

Fetch-and-add is also combinable with the other load/store primitives and still satisfy the paracomputer model. A write may be combined with a fetch-and-add,

returning the written value to the processor requesting the fetch-and-add, and forwarding a modified write to the memory unit.

The most interesting aspect of fetch-and-add is its use for process dispatch. Each processor in an ultracomputer which is available to execute a new task executes a fetch-and-add with $q = 1$ on a location in memory which is a pointer into an array of tasks to be executed. The returned value is a pointer into the array, and the memory location has now been updated to point to the new next task to be executed. Multiple processors may execute the fetch-and-add simultaneously, since each will get a different number back, each processor will get a separate task to execute. Each task will be executed on one and only one processor. This results in a considerable decrease in overhead compared to previous methods for software synchronization.

2.5. LISP machines

Of the commercial LISP machines available, there are four principal manufacturers; Symbolics, Inc., of Cambridge, Mass., Lisp Machine Inc., of Sunnyvale, CA. (now in receivership), Texas Instruments, and Xerox. The offerings of Symbolics, LMI, and Texas Instruments are somewhat compatible (source-code compatibility—they all use the dialect of LISP known as ZetaLISP).

The Symbolics, LMI, and Texas Instruments LISP machines all use an allocation of bits within a word known as “tagging”. Several bits (usually in the upper part of the word) are defined by the system designer to indicate special attributes of the data within the word. Tag bits are often used to indicate that the low order bits of the word contain a pointer, or an integer, etc. Tag bits are also used to indicate that a word has been visited in the current garbage collection cycle.

The Symbolics 3600 also uses tag bits to accelerate arithmetic. When the 3600 executes an **ADD** instruction, the adder immediately begins to perform an integer addition. Once the addition is underway, the microcode checks the tag bits of the operands. If both operands are integers, the addition is allowed to proceed. If either operand is not an integer, a microcode interrupt occurs and fixup software is invoked to perform type conversion. The net advantage in this scheme is to overlap the addition and the checking of tag information.

2.5.1. Xerox PARC

The Xerox products are the so-called D-machines. The machines (in order of increasing speed) are the Dandelion, Dolphin, and Dorado. All three machines were designed to be microcoded to the task at hand, rather than specifically designed to run LISP. Smalltalk and Mesa (a Pascal derivative) environments are also available, and in fact run considerably faster than the InterLISP environment. Xerox indicates that it takes about 200 microinstructions (at 60 nSec each) to execute a “call function” in Dorado InterLISP; hardly an encouraging statistic [Byte 1981], [Clark 1981], [Lampson 1981a], [Lampson 1981b].

2.5.2. M.I.T. CADR derivatives

The Lisp Machine Inc. hardware, the Texas Instruments hardware, and the older Symbolics hardware were based on the original MIT LISP Machine, a greatly altered PDP-10 called the CADR. In 1981 Symbolics redesigned the entire system for greater speed, at the cost of incompatibility of microcode. Symbolics has not offered a multiprocessor system. The Texas Instruments Explorer system supposedly can support multiple CPUs but no software exists to use multiple processors. Lisp Machine Inc. produces machines which are essentially four single-processor machines CADR's sharing a common disk system.

In a CADR-type machine, the simplest operation (not even a complete function execution) is the stack-pushing of a "small constant". Done by the microcode; this takes about 2 microseconds. Most instructions take longer, up to 30 microseconds [Verac], [Weinreb 1981].

The newer Symbolics offering (the 3600 family of machines) has a slightly wider microword than the CADR-based systems. The 3600 also supports tag bits in a more useful way than the CADR systems do [Moon 1987]. Instead of the microcode checking tags before an arithmetic operation is started, the operation is started and then the tags are checked. If the tags indicate that the operands are of the incorrect type, then a microcode trap occurs and a software fixup routine is entered.

In an exploratory effort, Steele and Sussman successfully produced a simplified LSI version of a CADR. This chip, called Scheme-79, did not support arithmetic functions beyond increment and decrement [Steele 1981].

2.6. Multiprocessor Lisp machines

In order to speed up program execution, several designs have been proposed which execute LISP in parallel. The method of detecting and utilizing parallelism varies between machines, but most designs use the applicative aspects of the LISP language to increase the likelihood of available parallelism. Once parallelism has been detected it is necessary to provide a means to allocate processors to the parallel programs to be executed.

2.6.1. Evlis

The multiprocessor allocation problem has been addressed to a small extent by the Osaka University Evlis machine. Evlis contains up to four processors which evaluate arguments to functions in parallel, on the assumption that evaluating one argument to a function will not affect the values of any other argument. A fifth processor performs I/O. The Evlis machine is based on the Intel I-3000 bit-slice chips [Yamamoto 1981].

2.6.2. MEF

MIT has proposed a research system for multiprocessor systems called the MEF (Multiprocessor Emulation Facility). The MEF is a network of 64 commercial LISP machines (Symbolics 3600 series) connected by an Ethernet-like cable. Because the entire operating system of the 3600 machine is an "open" system (there is no inter-task protection) any task may start or reference another task, including references via the network and network servers.

2.6.3. AMPS

A proposed machine by the University of Utah is the AMPS, or Applicative Multi-Processing System. This design uses a tree structure of processors. Internal nodes in the tree do not perform calculation, but are message routing nodes. Calculation is performed in the leaf nodes of the tree [Keller, 1979].

The internal nodes perform task allocation by means of a load managing algorithm. When a new task initiation request is created by a processing node, the message routing nodes move it throughout the tree to an area where little computation is taking place and therefore little contention will occur.

Because the intended software of the AMPS is applicative (applicative programming is discussed in the next chapter), each processing node has its own cache memory. There is no need for a cache invalidate signal, since no data element in an applicative system is ever altered once it has been written once. Unlike data-flow machines, the AMPS is demand-driven; a function is only evaluated if there is a need for the result. The AMPS uses a variant of LISP called FGL (Flow Graph Lisp).

The weakness of the AMPS design is that the tree network is subject to saturation as the tree grows deeper. If one level is added to the tree, the tree will have twice as many processor (leaf) nodes and the traffic through the root node will double. Thus, there is a fixed limit for any given technology in implementing an AMPS architecture.

2.6.4. AHR

Another applicative system proposed at the University of Mexico is the AHR. This design is similar to ZMOB in that it utilizes a number of Z-80's operating in parallel, but in this case, the Z-80's are executing an applicative variant of LISP. Evaluations are demand-driven, as in the AMPS, but a central memory store is used to satisfy requests for evaluations as well as to hold the results of the evaluations. This hardware, called the distributor, is the bottleneck in expanding the system without limit. The AHR was in operation in 1981 with 16 Z-80's [Guzman 1981].

2.7. Other Architectures

In attempting to gain performance, several new architectures have been proposed. These architectures do not fit neatly into the SIMD/MIMD taxonomy, and are discussed in this section.

2.7.1. The Warren Prolog Machine

D. H. Warren has proposed an alternate architecture for rapid execution of Prolog systems. The exceptional components of the Warren architecture are the large microstore, the Prolog-directed instruction set, the number of hardware-supported stacks, and the datapath oriented toward Prolog [Despain 1985], [Dobry 1985], [Tick 1983], [Nilsson 1984], [Warren 1983].

The most pertinent aspect of the Warren design with respect to LISP is a "parallel evolution" argument. The Warren design uses four stacks, one for dynamic storage of compound data items (the Heap), one for choice points and environments (the Stack), one for retaining binding/unbinding information (the Trail), and one for use as a scratchpad (the Push Down List). Instruction and data memory are separate and memory cycles can be run on both simultaneously.

As will be discussed later, partitioning of a single central stack into multiple special purpose stacks is a very powerful idea. By providing multiple physical hardware devices, the overall internal bandwidth is increased, and by requiring the localization of information of various types, the control system is greatly simplified. The applicability of the Warren approach will be seen in the DIS program-visible architecture. Although the DIS architecture does not use the Warren model of computation, and explicitly avoids the Warren microcoded control, any program which runs on the Warren architecture can be made to run easily on the DIS architecture.

A modification of the Warren architecture has been made commercially available as a two-board coprocessor from Xenologic, Inc, Newark, California.

2.7.2. VLIW Architectures

J. Ellis and J. Fisher have proposed an architecture called VLIW (Very Long Instruction Word) which has the potential for fast execution of non-vectorizable code. Their method (now being built by Multiflow Computer, Inc.) involves having a very wide instruction word with multiple fields, each field controlling a functional unit or register bank. All instruction fields in a given instruction execute simultaneously [Ellis 1986], [Fisher 1981], [Fisher 1984], [Fisher 1987].

The 1987 Multiflow offering could be ordered in configurations with one to four cardsets, each cardset having a 256-bit instruction. Each 256-bit instruction is partitioned into eight fields. Three fields control three integer arithmetic units, one field each are used for a floating-point unit, a register-to-register datamover, a register-to-memory datamover, a next-instruction-address (branch control unit), and a 32-bit field for "immediate data".

The major advantage of this design is that no interlocking of a pipeline is required by the hardware; the compiler can determine at compile time that a given register or ALU output is stable and valid. In this way, the compiler can determine proper pipelineing and can obtain significant parallelism even in non-vectorized FORTRAN code.

As a secondary consideration, the VLIW architecture permits compiler-generated rollback and rollforward of registers, so that when an operation is "optimized ahead" of a conditional branch, and the branch taken, the compiler has inserted a register rollback to re-obtain the unaltered state of the register. Likewise, when an operation is "optimized behind" a conditional jump, the operation is inserted into the code stream of both the branched and unbranched paths [Tjaden 1970], [Foster 1972], [Riseman 1972].

2.8. Summary of Current Architectures

We can summarize many of the computer design trends in a simple Venn diagram. The reader is referred to figure 2.8a . The open area in the center of the Venn diagram reasonably describes a multiprocessor DIS machine.

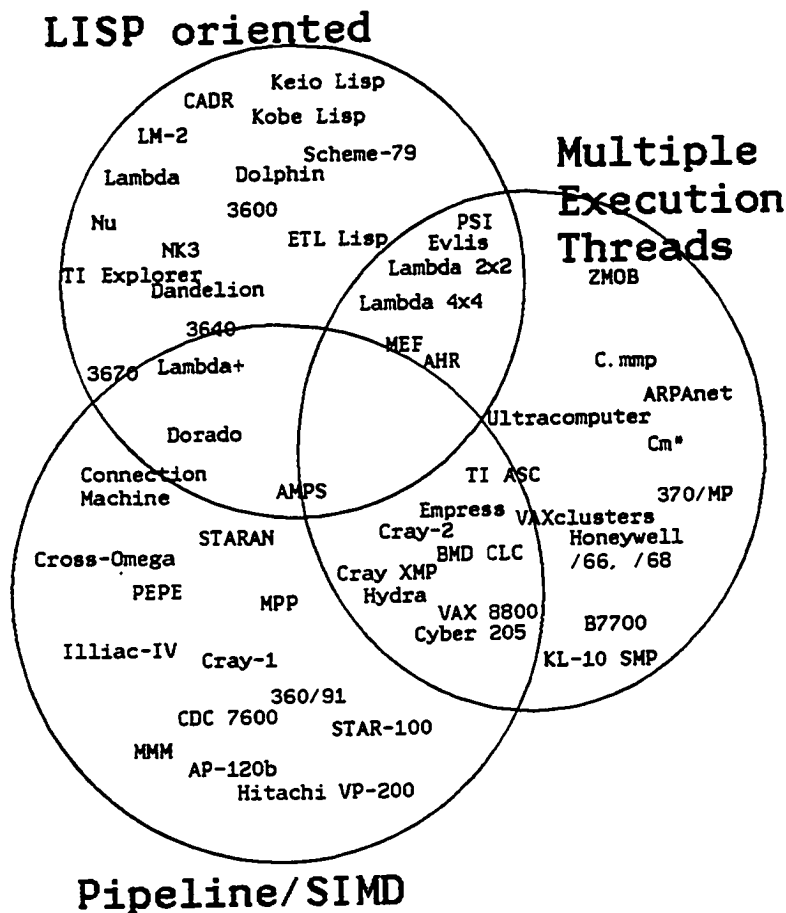


Fig 2.8a Venn Description of Computer Architecture

Current computer designs have significant weaknesses for high-speed computing. SIMD machines are not capable of efficiently executing conditional operations (because all conditional paths must be broadcast, the time to broadcast and execute the program is proportional to the sum of the lengths of the different individual control paths through the program). Current MIMD designs suffer from a lack of support software to automatically detect and utilize parallelism. Current pipelined machines cannot pipeline past a branch or conditional test.

Nearly all conventional serial computer designs have large amounts of state stored in the processor, which must be transferred to the memory on subroutine

call and restored on subroutine return. Vector machines also lack the ability to use the vector hardware in relation to any instruction (including subroutine call instructions).

The goal of this thesis is to show (by example and by simulation) that it is possible to execute LISP quickly with parallelism, to detect and use this parallelism automatically, and to thereby achieve "supercomputer" performance in the execution of LISP.

Chapter 3

Applicative Programming and Parallelism

3.1. Pure applicative programming

Pure applicative programming is based on the contract that a function shall execute and return a result without altering any memory location. The result is allowed to exist in some “alternate” area. Applicative functions must allocate their own local storage from a pool of unused memory and return all local storage to the unused pool on completion. Alterations to the locally-allocated pool are permitted in pure applicative programming, because the local pool is not accessible to any function except the one which performed the allocation.

Pure applicative programming is very convenient for multiprocessors, because of the applicative contract (no modifications to memory). Two applicative functions may execute in either serial order, or may be executed concurrently, and the results are guaranteed to be the same, because no modifications to memory have occurred. By a recursive argument, it can be shown that any number of applicative functions may execute concurrently without interference.

From a hardware standpoint, the applicative function contract has another convenient attribute; it is never necessary to invalidate a cache entry. Each processor can keep its own cache, under whatever cache discipline it chooses, and the values in the cache will never be different from the “current” values stored in main memory.

Applicative functions make the cache control strategy simpler, because there is never a “dirty” word in cache. Dirty cache words are words which have been changed by the central processor executing a store instruction; the cache has stored the new value, but the main memory has not been updated. Some cache strategies avoid this problem by using a “write-through” mode, where all writes to the cache immediately access main memory as well, but this misses a significant speedup that a “writeback” cache takes advantage of.

It should be noted that one can substitute the words “separate memory” for the words “separate cache” in the preceding argument without change in validity. Thus, an applicative language system is easily mapped onto a non-shared-memory multiprocessor without needing extra hardware to keep various memories synchronized.

3.2. Applicative Programming Variants

Pure applicative programming is impossible; the act of loading a program into memory involves making a change to memory. For this reason a looser interpretation of the applicative contract is usually taken: No function may alter an already allocated memory location. Unallocated locations may be modified once, at which point they become allocated locations and may not be modified. Local storage is drawn from a pool of temporary locations, and may be discarded at the function completion. A "garbage collection" system is invoked to gather up used temporary storage and make it available for reallocation as temporary storage.

This looser interpretation of applicative programming makes programming possible. Certain functions (such as the garbage collector) are permitted to violate the applicative contract and modify memory that has already been allocated. Function definition may redefine a function to have a new meaning (necessary for interactive debugging). Most of the functions in a system remain applicative; that is, remain side-effect free.

For an example, let us consider the difference between the statements

$A = B + C$

and

$B + C$

The first statement takes the value of B , adds to it the value of C , and then stores this in location A . Clearly this statement has side effects (modification of location A) and is not applicative. Any other program using location A must have some sort of synchronization contract with $A = B + C$ if update errors are to be avoided.

The second statement, $B + C$, only performs the addition of the value of B to the value of C . The function containing $B + C$ sees the result, but no other function can be affected. No synchronization contract is needed.

3.3. Applicative Result Passing

For the applicative statement $B + C$ to be useful, the result must be available to the calling function. This can be done by allocating an unused word, placing the result in this new word, and returning a pointer to the word, or it can be done by placing the the result on a stack. Both methods (and combinations of the methods) are useful.

The stack placement has several advantages over the memory placement. First, recursive functions need a stack-like construct for their local storage in any case. Second, removing a word from a stack returns the word to the free storage area automatically, a great advantage over garbage collection. Finally, if the function is written in such a way that it obtains its arguments from the stack (where they

were placed by previous functions) and places its result on the stack, then argument pass/result return is greatly facilitated.

3.4. Single-Assignment Programming

Similar to the applicative programming systems are the single-assignment programming systems. In a single-assignment system, no variable is written more than once. Under this contract, cache word invalidate is not needed. Pipeline processor design is also eased, because there is only a single small window where a value in memory does not have the "current" value. Instructions may be shuffled by the compiler freely, as long as all reads to X occur after the end of the assignment window of X.

Unfortunately, single-assignment languages cannot perform iterative loops. This deficiency, as well as the inability to reuse storage, has motivated the dataflow design.

3.5. Dataflow Programming

In a dataflow design, there is no instruction "sequence". It can be imagined that each variable is represented by a value calculator, and each value calculator waits for each of its arguments (which are the outputs of other value calculators) to complete their calculations and make a value available to the world. The synchronization contract here is that each value calculator must wait for each of its input arguments to be explicitly made available, before execution, and must wait for each value calculator which will use the current value to indicate that it has obtained a copy of the current value before it may process another value [Dennis 1984], [Gostelow 1980], [Rumbaugh 1977].

In implementation of data flow, each value calculator has a list of other value calculators which must be notified when a value becomes available. This eliminates "polling" of each value calculator. However, there is a heavy process-exchange overhead associated with dataflow processing whenever the number of possible calculations to be performed is greater than the number of available processors.

Dataflow systems have another disadvantage- ALL values which are calculable in a given problem are calculated, rather than just the ones that are necessary. Those values which are not needed are thrown away. It would seem preferable to have some method of directing the course of a calculation other than explicitly switching the writing of an output variable to one of several reading functions.

3.6. Loosely Applicative Languages

There are several loosely applicative languages in common use today. For real-world use, the applicative contract is usually not rigidly enforced; any program may contain or execute a function with a side effect. Loosely-applicative languages in common use today include LISP and APL.

In LISP and APL, the syntax of the language and the system-supplied functions encourage applicative programming. Both languages support the concept of a function result being the immediate argument to another function, without being explicitly stored into a memory location. In contrast, FORTRAN programming discourages applicative style; there is no way for one statement to pass a data item on to another statement except via an assignment (which places the data item into a fixed (therefore probably re-used) memory location).

LISP provides the programmer the choice of whether or not to use an applicative or non-applicative form of a function in many cases. As an example, LISP provides two functions to reverse the order of entries in a list. `Reverse` allocates new memory to hold the reversed list while `nreverse` (the non-applicative form of `reverse`) reverses the list "in place", destroying the old list but hopefully saving memory.

3.6.1. Applicative LISP

In general, a function in LISP is side-effect-free if it calls no function with a side effect. LISP does not distinguish between user-supplied and system-supplied functions, so this definition is sufficient to determine whether or not any function in a LISP environment is side-effect-free.

A LISP function which may or may not cause a permanent modification to memory is not be considered to be side-effect-free. Data dependency is not considered in determining whether or not a function is side-effect-free (with the exception for the pathological case of a data-dependent error which prints out a message is not considered a side effect).

3.6.2. Non-Applicative LISP Functions

There are some LISP functions which only exist in the non-applicative form. The significant non-applicative LISP functions are `defun`, `putd`, `put`, `putprop`, `rplaca`, `rplacd`, `store`, `setq`, `makunbound` and the I/O functions. `Defun` and `Putd` create or change the definition of a function (`defun` actually calls `putd`). `Rplaca` and `Rplacd` replace the head and tail of a list with a new value. `Setq` permanently changes the value of a variable (until the next `setq`, that is). `Store` permanently changes the value of an element of an array. `Put` and `putprop` place a piece of information about a variable on the the variable's property list- this is different information than the value of a variable. `Makunbound` erases the existence of a variable. Finally, the I/O functions are all non-applicative, because from the point of view of the user,

execution of an I/O function changes blanks on the paper or in a file to character output.

3.7. Computability of Applicativeness

We can determine for a given set of LISP functions forming a program which functions are applicative and which are not. This can be done by:

1) Search the text of input functions, obtaining the name of every function defined. Assume (for now) that all the user functions are applicative. Call this the Assumed Applicative list, or, for short, the AA list.

2) For each user function defined, make a list of every function it explicitly calls, ignoring data dependent function calling.

3) Obtain a list of non-applicative system-supplied functions (perhaps from a permanent file). Call this list the known-non-applicative list, or KNA list for short.

4) For each function on the AA list:

IF the function calls any routine on the KNA list:

4.1) remove the function from the AA list.

4.2) place the function on the KNA list.

5) IF in step 4 we removed any function from AA, goto step 4.

6) Done. All functions named in AA are applicative.

This algorithm may not be optimal, but it is guaranteed to terminate for a finite set of input functions. This termination is forced because the list AA must get at least one element smaller every time step 4 is executed, or else the program terminates immediately. Since AA is finite and decreasing with every execution of step 4, the program must terminate.

Once we know what functions in a LISP program are applicative, we may use this information opportunistically at runtime to provide parallelism. For example:

```
for I = 1 to 1000 step 1
  <any-applicative-function>
  <another-applicative-function>
  <yet-another-applicative-function>
next I
```

can be subjected to the applicative-determination algorithm, and found to have 3000-fold parallelism, exclusive of any parallelism that might lie inside each of the

three called functions. The reason for using LISP instead of FORTRAN is that it is nearly impossible to write applicative FORTRAN functions, but most LISP functions are applicative, with no special effort made by the programmer.

3.8. Costs of Applicative Programming

Applicative programming on general-purpose machines has been hampered by the style in which applicative programming has been promoted. In particular, there is a tendency to make every function redefineable, including such primitives as "addition" and "equality". Allowing redefinition of these functions means that the compiler for an applicative language must not insert inline code; instead, it must search a symbol table for the current definition bindings of each operation, including primitive operations.

This searching essentially eliminates the possible optimizations allowed by inline coding, and places the procedure-call instruction as the most common instruction in the program. Since procedure call is also usually one of the slowest instructions to execute, applicative languages have gained a reputation for slowness. Common LISP attacks this problem by specifically declaring that certain primitive functions will be compiled "inline", unless a redefinition of them is seen by the compiler lexically before the actual call. This method gives some speedup but greatly reduces flexibility.

It is a goal of this thesis to construct a computer architecture which has a very low (if possible, zero) overhead for procedure calling, and can therefore execute LISP and other applicative languages at high speed.

Chapter 4

The LISP Language

This section is concerned with the syntax and style of the LISP language. The syntax of LISP is based upon Church's lambda calculus, and although it is necessary to have some understanding of the syntax of LISP, it is the style generated by the syntax of LISP that causes a large proportion of the functions to be applicative.

LISP, like many other languages, has several dialects. The **Franz LISP** dialect is described in [Foderaro 1981]. The dialect we will use for the majority of this work is **Common LISP**. A complete description of Common LISP may be found in [Steele 1984].

4.1. Parenthesis

The first thing which strikes a non-LISP programmer when seeing LISP is "look at all those parentheses!". The parentheses in LISP specify groupings of one operator with its operands. For example:

```
( + 2 3 )           ; everything following a
                    ; semicolon is a comment
                    ;
                    ; --> 5
```

4.1.1 Prefix Notation

LISP functions may have any number of arguments (including zero arguments). In LISP, the function name is the first element of a list, and each element of the list following the first is an argument to the function (prefix notation). One function may be evaluated to give the result which is immediately passed on as an argument to another function. Because of the prefix notation and the pass-back ability it is common to write:

```
( +
    ( * 2 3 )           ; intermediate result of 6
    ( + 1 1 )           ; intermediate result of 2
)                       ; --> 8
```

4.1.2. Argument Passing

Arguments to LISP functions are passed as defined by Church; what is passed is a copy of the actual argument. The called function may modify the local argument freely; the caller has its own copy which is not modified. If a called function performs some non-applicative function on the argument, like following a pointer and then altering the location pointed to, then the caller may see the altered value. This type of variable passing is called lambda-binding.

We should note that this form of argument-passing maintains the applicative-programming contract, even if the called function performs a non-applicative operation on an operand (but not on an object pointed to by an operand).

4.1.3. Program Representation

LISP is different from most other programming languages in that LISP programs are represented in the same way as LISP data, namely, as lists. The internal representation of `(+ 2 3)` is the same for noncompiled code and for data. It is possible to call the LISP evaluator function on a data list; as long as the first element in the data list is the name of a function which has been defined somewhere, the evaluator will return the same value as if the list had been typed in at the keyboard.

4.2. Variable Scoping

Lambda-bound variables have certain difficult attributes in a compiled-code system; this is often called the variable-scoping problem. In classical interpretive LISPs, a variable referenced in a function which did not lambda-bind that variable gets the value bound to that variable in the calling function. If the calling function did not bind the variable, then it gets the value bound in the caller of the caller. This procedure continues until the top level is reached. If the variable was not bound (by a `setq`) at the top level, then an error is generated.

An example is in order here. We will set the value of global variable `X`, print `X`, bind `X` (with a lambda-binding), print `X`, unbind `X`, and print `X` again.

```
(setq x 99)                ; X now is 99

(print x)

99

(defun test-bind (x)        ; define our test function
  (print x)                 ; which lambda-binds X
  (setq x 'forty-two)
  (print x))

(test-bind 'test-val)      ; run the test function
```

```

test-val
forty-two           ;results as expected

(print x)

99                 ;and the old value
                  ;has been restored.

```

Compiled-code LISP systems usually do not support this backward searching for lambda-bound variables. The difficulty is that binding and unbinding the variable on every call and return takes a lot of time, and in most cases, the lambda-bound variable is only used locally, so the effort of saving and restoring the lambda-bound variable in the global table is totally wasted. For this reason, most compiled-code LISPs (including LISP Machine LISP and Common LISP) specifically do NOT allow a lambda-bound value to propagate down a call tree.

This non-dynamic scoping of lambda-bindings would cause great compatibility problems, so a compromise route is taken in most compiled LISP systems. By default, a variable binding is not propagated to the called functions, which makes the code run quickly. If the user really needs to propagate a variable to all callees, he may declare the variable "special". Special variables are known to the compiler, and the code which references special variables is intentionally modified to chase the call tree and obtain the correct value.

By allowing a "special" declaration, the compiled LISP systems maintain fast code in the great majority of cases, and allow compatibility in the few cases where a variable really does need to be passed down the call sequence. Some LISP compilers can automatically detect that a variable probably should have been declared special (but not always in time - it may be necessary to recompile certain functions because of a concealed special requirement). It is generally accepted that the compiler should warn the user of such "automatic special" declarations, because the automatic special declaration is usually a symptom of a typographic error, rather than an intentional variable passing.

In terms of hardware, the choice becomes whether to keep only a stack of bound values (called deep binding), or to keep a list of current values and a stack of values-to-be-restored (called shallow binding). Deep binding binds and unbinds in constant time, while searches take time linear with the number of bindings done. Shallow binding searches in constant time, while binding and unbinding may take linear time (if a non-nested binding form is allowed). Hence, most InterLISP implementations use deep binding (because the InterLISP spaghetti stack allows non-nested binding forms). Common LISP implementations generally use shallow binding, because the Common LISP PROGV function generates strictly nested forms.

4.3. Functions as Arguments

Because LISP functions are stored as LISP data, it is possible to write a function which takes another function as an argument. This provides a succinct way of indicating iteration:

```
(dotimes i 100 ( print i ) )
```

prints the integers from 0 to 99.

LISP provides an iterative facility based on lists instead of arrays:

```
(mapcar
  (lambda (x) ( + 1 x ) )
  list-of-numbers
)
```

returns a list of numbers, where each number in the returned list is equal to one plus the number in the corresponding position of the input list-of-numbers. There is no need to know how many numbers are in list-of-numbers; any number of elements, including zero, is permissible. The lambda function may also be a function of more than one variable:

```
(mapcar
  (lambda (x y) (* x y))
  first-list
  second-list
)
```

which returns a list of numbers, each element in the list being the product of the corresponding elements in first-list and second-list.

The lambda-definition in a mapcar may be replaced with any function, by using the "getd" function. Getd of a function returns the function's lambda-definition:

```
(defun my-product (x1 x2)
  ( * x1 x2 ))

(mapcar
  (getd 'my-product)
  first-list
  second-list
)
```

which is equivalent to the previous example.

LISP provides more general operators than the "do" and "mapcar" used as examples. In particular, any user-defined function may be used to obtain the next element of a list to be processed by a map function, and any user defined function may be used to define the start value, next value, and termination test of a do function.

4.4 An Abstract LISP Engine Design

With the above knowledge of LISP, we can now consider an abstract model of how a LISP supercomputer ought to look.

4.4.1. Data Needs of LISP

Because LISP is applicative, we know that a LISP machine ought to provide direct support for applicative function calling and return. Stacks provide this kind of support for applicative languages. We should therefore expect a stack-intensive rather than memory-intensive architecture.

Because the applicative contract applies to machine instructions, we expect that a separate program-storage memory may exist.

Because LISP is more a language for pointer manipulation than a language for arithmetic, we will need the ability to follow a pointer quickly (perhaps even the ability to pipeline pointer following), instead of the ability to do arithmetic quickly. Since it rarely makes sense to dissect various parts of a pointer (unlike floating-point numbers, with separate mantissa and exponent), we expect a word-oriented rather than a byte-oriented design.

Because LISP is call-intensive, we should expect a minimum of context to be saved and restored during a procedure call.

4.4.2. Needs for Parallelism

If we are going to evaluate LISP with parallelism, we need the raw ability to execute different operations at the same time. We can achieve this easily by having multiple functional units within a single processor.

We will expect a flexible inter-unit data communications system, so that when several units need to communicate, the chance is small that a restriction in the datapath will keep those units from communicating.

Since we are hoping to execute multiple operations in parallel, we should expect to see either a deep pipeline, with a large control structure (microcode), or the ability for the compiler to generate, at compile time, very complex instructions that choreograph multiple unit operations.

The only "choice point" available in the above abstract design is whether to use a complex microcode to control the multiple functional units and data paths, or whether to allow the compiler to configure and optimize them at compile time. Considering that the compiler and microcode are both programs, either one *could* perform any optimization that the other could perform *if it had sufficient time*.

Unfortunately, microcode doesn't have a lot of time available. The compiler can spend as long as it takes (minutes, if necessary) trying different arrangements of

instructions in an effort to find the fastest alternative. The microcode will have at most a few hundred nanoseconds. For this reason, we choose to have a very wide instruction word, with a very smart (and slow) compiler.

4.4.3. The Abstract LISP Supercomputer - DIS

Assimilating the above requirements, we can envision what a LISP supercomputer should look like. This abstract model comes very close to the actual DIS design (see chapter 7). To summarize, we expect:

- Multiple Functional Units
- Stack-Oriented
- Separate Memories for Instructions and Data
- Memories designed for Pointer Following
- Wide, Flexible Datapath Interconnections
- Ability to control all of the above simultaneously
- Weak (or no) Floating Point

The DIS architecture exhibits all of these features.

Chapter 5

System Design Constraints

This section deals with the design constraints on this project. Some of these constraints are "essential" in that they are requirements for any supercomputer design, and some are technology or previous work constraints.

5.1. Uniprocessor/Multiprocessor

A uniprocessor configuration should be able to do useful work

Even if the design is a multiprocessor design, it must be capable of useful work with only a single processor. We wish to solve the problem of previous designs concerning uniprocessor speed. Our proposed solution is that the addition or subtraction of processors should be transparent to the software, and should only affect speed of execution. Clearly, a problem which runs on a multiprocessor but does not run on a uniprocessor violates this concept.

5.2. Maximum Processors

No artificial limit on the number of processors

The maximum number of processors in a system should be upwards of one million. Many problems currently under consideration have 1000 x 1000 arrays of information. Because we wish to take advantage of the applicative nature of LISP to obtain parallelism, we should be able to include sufficient processors to allocate one processor per data element.

5.3. Language Orientation

A LISP-oriented architecture

The design will be oriented toward applicative languages such as LISP. This constraint is due to previous work and commercial products which use non-applicative language models. It is a contention of this thesis that an applicative model of processing provides a great opportunity for parallelism and therefore greater speed.

5.4. Package Pinout

No limitation on package pinout

The number of connections within a single processor is specifically not limited. Packaging technology is currently limited by cost rather than ability. 145-pin IC's are in limited use, and there is no reason to believe that 512-pin IC's are unachievable.

5.5. Gate Delay Speed

Use commercially available part speeds

The technology used to implement the processor is specifically limited to the speed of currently available parts. Although an actual fabrication may be in mixed NMOS, CMOS, TTL, and ECL, we will not assume any speeds which have not been achieved in the fastest parts available in the marketplace.

5.6. Network Interconnect

Use a standard network interconnect

We will take a network interconnect from the published literature [Agrawal 1983], [Bhuyan 1983], [Bhuyan 1984], [Chu 1984], [Gonzalez 1972], [Marsan 1983], [Oruc 1984a, 1984b], [Von Conta 1983]. The communications network itself is not part of this design.

Chapter 6

Design Methodology

This section is concerned with the methodology to be employed in the design of an applicative supercomputer. Commonly held considerations are discussed, and the preliminary results are discussed.

6.1. Ranking Methods

A major difficulty in the area of computer design is that there are no strong theoretical bases to assert one design is "better" than another. After a design has been in use, some metrics may be applied to gain a qualitative ranking (which is often based on the software generated for the architecture), but these are employable only after the design has been built and made available for users to program. This design will leave the qualitative measurement of computer design as an open question.

Some authors have based appraisal of computer architectures on the presence of desirable "features" (desirability being in the mind of the author). This approach has been validly questioned by the Reduced Instruction Set Computer promoters, who show that the presence of special instructions and features may be counterproductive to increased throughput [Ponder 1983].

6.2. Orthogonality versus Preferentiality

Despite the "black art" nature of computer design, several concepts have emerged as worthy of discussion. Many machines are designed with the concept of "orthogonality" in the instruction set. This means that if an option, such as indirect addressing, is available on one instruction, the same option should be available on all other instructions. Likewise, there should be no "preferred" registers; every register is usable as an operand for any instruction.

Orthogonality in a design can lead to large instruction sets and large microcode loads. Unfortunately, the flexibility of a completely orthogonal instruction set is usually unused in a compiled language, because the code emitter generally dedicates certain registers to certain operations. A superior optimizing compiler might use the extra registers, but it is unclear how many optimizers take advantage of this. The typical peephole optimizer cannot utilize extra registers in most cases.

There are machines with un-orthogonal instruction sets; the Intel 8086 series CPUs have several preferred registers. The advantages of a preferred register architecture include a more compact instruction set, and a smaller microcode (less likely to contain bugs). By coercing the compiler writer to use the preferred registers in the designed-for manner, the CPU designer and microcoder can arrange the internal data connections to give greater speed in most cases.

There should be strong consideration of the amount of information stored with the CPU. Although CPU-local information may be the fastest to access, it must be stored on every context-switch operation, including subroutine call. The VAX is particularly prone to this problem; there is over half a kilobit of user-visible registers alone to be saved and restored on every subroutine call in a VAX CPU.

6.3. Iterative Design

Although the author is unaware of any firm theoretical basis for computer design, there is an informal method based on compatibility and iteration. If the computer is to be compatible with previous models, then the user-visible architecture will not change at all. If the computer is a "new line", then the design is arbitrary. After completing the arbitrary design, create a simulator, and write programs to run on the simulator. As programming weaknesses show up, modify the design and the simulator. The process terminates when either time or money is exhausted.

This iterative process for design of a computer implies the use of another computer to execute the simulation. Some designs have been simulated on paper "computers" rather than electronic computers; the design proposed in this thesis exists only as a large (5000+ line) set of VAX Common LISP macros and functions. One of the tasks in fully developing this design has been the production of a simulation system accurate and flexible enough for testing purposes.

This iterative process does lead to a number of different designs being given attention. It also means that a majority of the design effort is expended on designs which will not be used. Since the final design is evolutionary rather than revolutionary, there will be vestigial features present which are unnecessary and contribute to the final cost and complexity without providing significant gain.

6.4. Functionality Provided

This design is directed toward LISP execution, so it is reasonable to provide a significant portion of the basic LISP functions in hardware (or nearly hardware). The LISP 1.5 definition requires only 6 functions in addition to the purely mathematical primitives such as `add`. The six list functions are `car`, `cdr`, `cons`, `eq`, `atom`, and some form of conditional.

The `car` function returns the head element of a list. The `cdr` function returns all but the head element. `cons` returns a newly allocated storage cell whose first pointer points to the first argument, and whose second pointer points to the second element. `eq` compares two pointers, and returns "true" if they point to the same storage cell, else `eq` returns "nil". `atom` returns "true" if its argument is an atom, else `atom` returns "nil". A conditional can be expressed in several ways, but the most common is as consecutive if-then-elses.

Given these primitives, a fairly complete (although inefficient) LISP can be built. Since we are designing a Common LISP capability, we target our efforts toward an

architecture which can execute the above primitives in a few microcycles each. Other LISP primitives are also to be considered for fast implementation, but it has become experimentally clear that a design which executes the six primitives poorly will have difficulty with the more abstract functions as well.

6.5. Final Comments on the Design Process

Besides the simulator, system support software has been written and tested. The support software includes a recursive-descent LISP compiler, an assembly-language level optimizer, an assembler, and a linker/loader. Heavy use is made of LISP macros (in fact, macro-defining macros) to keep the design self-consistent, even with a number of persons simultaneously working on various aspects of the system.

At this point it is reasonable to mention the members of the DIS design team. I (William Yerazunis) did the overall design and the core system-definition macros. John Tribble, Eric Luce and I corroborated on constructing the DIS simulator. John also did the graphic monitor for simulator observation. Paul Charlton did buildability analysis for most of the DIS functional units. Randall Shane and I split the work on the DIS Common LISP compiler, he working on a full-blown multiprocessor compiler and I a simpler uniprocessor compiler (which could be completed more quickly).

Chapter 7

The DIS Processor Design

We will now describe the DIS (for Direct Instruction Sequencer design. The DIS design cleanly satisfies the requirements of a fast subroutine call/return and fast execution of LISP primitives, as well as fast servicing of local variables, special variables and variables bound through the bind function. The facility for multiprocessing will probably be modified in the future to speed up the passing of special variables. The issue of compiler-generated calls to the multiprocessing facility is currently being considered separately by Randall Shane.

A simplified drawing of the DIS architecture is shown in figure 7.0a below. The vertical lines are distribution busses, the dark horizontal lines are distribution bus drive, and the light horizontal lines are distribution bus receive. Each box represents a separate functional unit, described below.

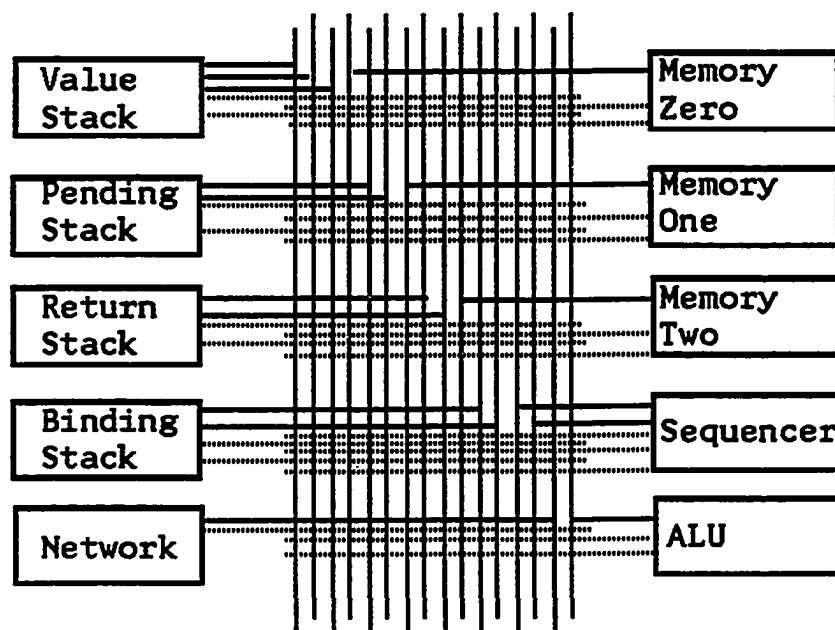


Fig 7.0a A Single DIS Processor

The individual processor is organized as a multiple bus machine; each of several modules drives its output onto a distribution bus, for access by any other module. Modules also accept data from distribution busses for internal use. The actions

of the modules in the system are controlled by the sequencer, which sends control signals determining what busses will be read by what modules, and what actions will be taken by that module.

Perhaps it is misleading to call the distribution bus system a "bus" system, since only one module will ever write data to any particular bus. This configuration of dedicated distribution busses does give us a significant speed advantage in that there is never any need to run a bus arbitration cycle, nor are there any wait states needed for a functional unit to obtain a needed bus.

Because of the wide instruction word (256 bits) each module is controlled by dedicated bits in the instruction word. The sequencer has no need for an instruction decoder because command bits to one module do not depend on command bits to another. Essentially, the instruction word is stored in the decoded form in memory.

It is from this idea of direct storage of an unencoded instruction word that the acronym DIS is formed- DIS stands for Direct Instruction Sequencer. By not encoding the instruction, we retain complete flexibility to control each functional unit (stack, memory, ALU, next-address-generator) independently of other actions occurring in the same instruction cycle. The control loop is now to read a bus, broadcast the bits to the appropriate functional units (which can be done with wires, as the bit location defines exactly where that particular instruction bit is to go), wait for all modules to indicate completion, and repeat.

DIS is also a pun on "The Inferno", by Dante Aligheiri, as the name of the Fortress of Hell, whose ramparts are made of red-hot iron [Aligheiri 1315]. The concept of red-hot, glowing iron is somehow strangely attractive when one considers designs for supercomputer systems.

Storing the unencoded form of an instruction is not new. A. M. Turing proposed a very similar idea for his design of the Mark II in the early 1950's [Hodges 1983]. It was discarded then because the cost of memory was high compared to the cost of hardwired controllers. Since it appears that the cost of memory will continue to decrease, we have adopted the idea of storing direct instructions for this design.

The DIS machine is not designed to operate as a standalone processor. It is expected that a parallel interface (probably 32-bit) will exist between a uniprocessor DIS machine's Network Interface and a host processor, such as a MicroVAX Workstation or Sun. The host processor will be responsible for initializing and loading the DIS machine, for servicing all I/O, and for serving all interrupts.

7.1. Basic Modules

The basic modules of a DIS machine are the local memories (M0, M1 and M2), the value stack (VS), the ALU, the pending call stack (PS), the return call stack (RS), the binding stack (BS), the network interface (NI), the operation sequencer (SEQ), and the optional input/output unit (IOU),

In the simulator, no input/output is allowed, the network interface is malleable, and the debugger is a set of user-invoked LISP functions.

Because some operations in certain modules (such as a page fault in memory or a floating point divide) may take more time than the nominal 100 nanosecond cycle time, two synchronization signals are provided. The first signal, which is pulled down by every module which has not completed the current operation, is led to the sequencer. After being masked by logic bits in the current instruction (such as ignore a module, such as the ALU, not being ready), the ready bits are WIRE-OR'd. This signal, MODULE-WAIT, is read by the sequencer logic and inhibits the sequencer from moving to the next instruction. When all modules have completed the operation, MODULE-WAIT goes high and the sequencer proceeds.

The second synchronization signal is GO. GO is asserted high. All modules latch their command inputs on the rising edge of GO, and proceed to execute their respective commands. GO is driven out by the sequencer.

It should be noted that all devices with more than a few bits of state information have "hidden" connections to system memory. All stacks and memory ports reference local memory, which is partitioned into sixteen areas. Each memory-using device nominally accesses only one area, and therefore no cache-invalidation signals are passed. A reference across memory partitions causes the appropriate caches to be written back before the reference is allowed to proceed. The cross-memory reference access happens only under program error recovery conditions and does not occur in normal operation. Cache-invalidate is passed as a one-bit connection within a module.

This hidden interconnect is called the "back door bus", and it is accepted that whenever the a cross-module reference occurs the machine will spend a cycle stalling while the addressed memory performs the extra cycles necessary. The current compiler never generates any such back-door bus references.

Note: In the DIS simulator, it is assumed that cache always misses for timing calculations. Alternately, one can view the simulator as a machine which has had cache disabled and therefore has a constant, relatively slow memory cycle time.

System memory partitions may use a virtual translation mechanism to allow the use of rotating mass storage on a processor. This virtual translation mechanism has a private connection to the IOU. Because the IOU is optional, in some cases the virtual translation mechanism is able to provide relocation but not virtual memory. In the current simulator, I/O is forbidden, and therefore virtual memory paging never occurs.

The choice to ignore the virtual memory question in this simulation is not unreasonable. For a discussion of the advantages of an architecture which never pages because of a great amount of memory, see the MMM (Massive Memory Machine) [Garcia-Molina 1984].

7.1.1. Memory

There are three functional units which use memory in the von Neumann style. These memory units are designated M0, M1, and M2. Each accepts a 64-bit address and returns 256 bits of information. The unused higher-order bits of address are used to contain tags and global-processor addresses. Tag bits are ignored by memories; global processor addresses are available but the uniprocessor simulator ignores them.

M0, M1, and M2 are physically similar devices. The current compiler uses them differently, so they appear to support different options, but in fact they are logically and functionally identical cards.

The four memory operations are Idle, Read, Write, and Fetch-and-Add. Each memory must perform one of these operations per cycle. Each memory has the capability to read two distribution busses, one for address, one for data, as well as accepting a three-bit literal field from the instruction which is used as an offset from the bus address. The internal data paths needed for these operations are shown in figure 7.1.1a

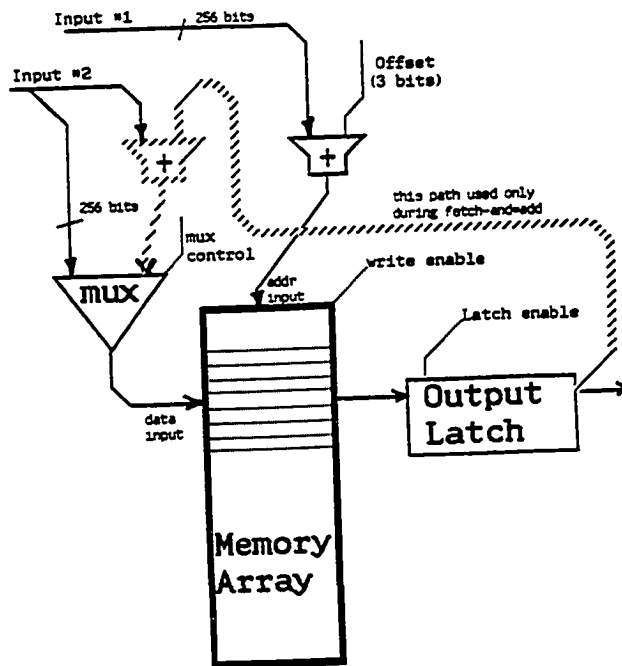


Fig. 7.1.1a A DIS Memory Unit

Idle causes exactly nothing to happen to the value latched in the output port of a memory unit. The previous value remains there, and will be readable by any unit reading the distribution bus attached to that memory unit. The purpose of the

Idle operation is to prevent a possibly costly cache invalidate, partition contention, or even a useless page fault.

Read causes the memory to accept an address, add to it a three-bit offset (a literal given in the instruction word), and set its output port to the value at that memory location.

Write causes the memory unit to accept an address and a 256-bit datum word. The three-bit literal offset is added to the address, and the datum word is written to the addressed memory location. The memory output port is also given a copy of the datum word.

Fetch-and-add causes the memory to accept an address and a value. The memory adds the three-bit literal offset to the address, and fetches that datum word. The datum word is placed in the output port. The value is then added to the datum word, and the result stored back into the original location. Note: the current simulator supports fetch-and-add; however, the current compiler never generates that instruction.

M0 is used only for instruction fetches, and therefore the compiler generates only READ operations on M0. M0 usually reads the 64 bit address from the sequencer Next Address output bus. M1 is used by the current compiler for holding the shallow-binding symbol table. M2 is used by the current compiler to hold all other information, such as conses, arrays, strings, etc.

The reader should be aware that no hardware restriction prevents M0, M1, and M2 from being used interchangeably. Indeed, R. Shane uses a slightly different mapping of use-to-device in parts of the multiprocessor allocation system [Shane 1987].

7.1.2. Stacks

There are four main stacks in the DIS design. These stacks are the value stack, the pending frame stack, the return frame stack, and the binding frame stack. Each stack nominally accesses only one partition of memory. To increase the bandwidth of the stacks, the stack memories are configured with odd/even address interleaving. In this way, the top two elements (rather than just the top element) of each stack is visible to other functional units.

Because no direct access to the stack pointers for the binding, pending, and return stacks is needed during normal operation, these stack pointers are referenced via multiplexing onto the same distribution bus used to distribute the instruction literal data. A two-bit field in the instruction controls which of the four possible values will be placed on the bus.

Like the memory units, the four stack units are identical physical cards, with inputs and outputs rerouted. No hardware restriction exists to prevent interchange

of use nor additional uses being placed on a given stack. Figure 7.1.2a shows the anatomy of a stack unit.

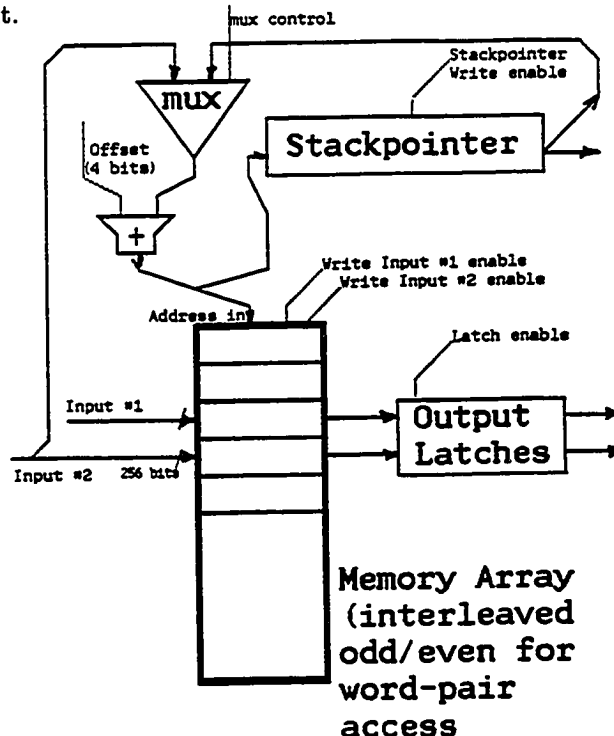


Fig 7.1.2a A DIS stack unit

Each stack unit executes at most four operations in one cycle. Valid operations are stack idling, pointer selection, pointer modification, value writing, and pointer writing. Each stack unit also has a pair of distribution bus inputs and a pair of distribution bus outputs.

In the first phase, a stack pointer is selected from one of two sources; either the internal (saved) stackpointer is used, or one of the two input busses is selected and the datum from that bus is used as a stackpointer. Two bits is used here- the extra state is used to force a stack idle.

In the second phase, a four-bit displaced-by-eight offset is added to the selected stackpointer. This generates a new effective stackpointer. Four bits are needed in this phase. The choice of four bits in the displacement was somewhat arbitrary, and is justified only on the grounds that functions of more than seven arguments were, in the author's experience, rare.

In the third phase, zero, one or two words may be written onto the location pointed to by the new effective stackpointer and one less than that location. Thus, we can overwrite the new top of stack, overwrite one below the new top of stack, overwrite both the top of the stack and one below the top, or overwrite neither.

This level of control requires two bits. It is envisioned that memory be interleaved at least two ways in order to support this writing of two words in one cycle.

In the fourth phase, the new effective stackpointer is optionally written into the stack unit's stackpointer register.

7.1.3. DIS Compiler Stack Usage Model

The model the DIS compiler uses to control stack usage directly influenced the names of the four stack units. Each stack unit is used for exactly one class of datum, so that the recursive-descent compiler can (in general) not need to know anything from previous levels of recursion.

The stack units are permitted to make back-door-bus references, but the current compiler does not generate such instructions.

7.1.3.1. Value Stack Usage

The value stack holds data or pointers to data. This data might be being manipulated, or it might be an argument being passed to a function. Results being returned from a function are also placed on the value stack.

While within a called function, the local variables (in the argument list) may be quickly referenced by directing the value stack to use a saved argument pointer as a temporary stack pointer, and indexing from that value with the four-bit offset field. While only the first seven arguments can be read this way, most functions have fewer than seven arguments and we save one cycle versus a memory oriented system. If there are more than seven arguments, the first seven are accessed in this fast manner and the remaining arguments require an ALU cycle (to calculate an address) followed by the stack unit cycle for access (total extra cost: one cycle). Alterations to locally bound variables (SETQ's to lambda-bound arguments) are handled similarly, by writing with an offsetted stack pointer.

7.1.3.2. Pending Stack Usage

The pending and return stacks work together to speed subroutine calling and return. In general, the pending stack holds frames of calls whose argument lists are being built, and the return stack holds frames of calls or conditionals who are evaluating their argument lists.

A frame on the Pending stack consists of two words. The upper word is a pointer to the first instruction of the function being called. The lower word of the frame is a saved copy of the value stack pointer BEFORE the first argument to the function was pushed. This saved pointer is used inside the called function as a base for local variable reference, and is also used to quickly unwind the stack when a function returns a value.

The interaction of the Pending and Return stacks with the Sequencer during a function call are most easily seen in Fig 7.1.3.2a. In this figure, a call frame has been created and is being executed. Control is transferring to the first instruction of the called routine, and the necessary return information (return address, old frame pointer) are being saved on the return stack.

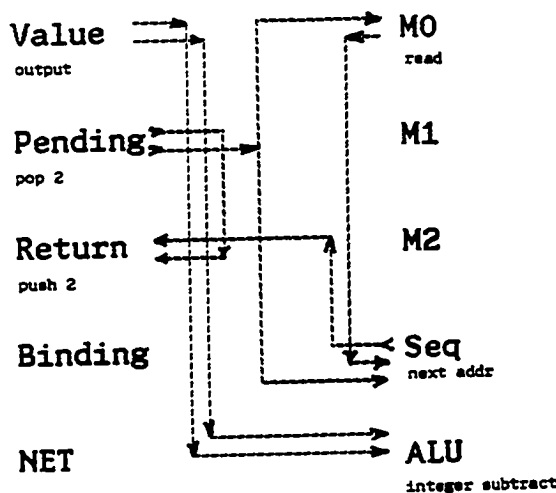


Fig 7.1.3.2a Data Flow during Function Call

An obvious question is "Why save the location to be called on a stack, when it could be supplied as immediate data?" The answer to this is simple: bandwidth. We can only get one immediate datum into the machine from the instruction stream per cycle. If a function is being invoked on a constant, then we need two cycles of instruction just to get necessary literals into the data path (one literal for function address, one for the constant).

We have two choices- first send the function address, then the data, or first the data, then the function. In a strictly von Neumann machine, it would not make any difference. Because M0 (the instruction memory) has a visible input and output, we need to give the address of the function to M0 two cycles before the first instruction of the function can be distributed to the functional units. If the datum is a fixed constant (like a number, or an immediate pointer), we need have it available to be placed on the value stack only one cycle before the first instruction of function is distributed.

The net result is that the DIS machine can go 50 % faster (two cycles instead of three) in such cases, by prestacking both the call target as well as the argument pointer. The cost is a doubling of the memory usage to hold pending call pointers.

Once the pending call frame has been constructed (by pushing of two pointers), the arguments to the function are pushed onto the value stack. In the case of literals and symbols, this is done directly by inline code. Certain functional forms are also

executed by inline code (such forms as `cond`, `setq`, integer mathematics, basic list manipulation, etc). If a form is compiled which is not one of these inline-coded forms, then the compiler recurses on that form.

Eventually, the argument list is complete and the called function is invoked. First, the M0 memory is instructed to use the top value of the pending stack as a pointer to the next instruction. Since the current instruction stream is to be returned to, we must save the next instruction address (generated by the sequencer), and the argument frame pointer (held on the pending stack), by pushing these values onto the return stack. Simultaneously, the pending stack is popped twice, discarding the pending frame, and the sequencer itself is directed to use the address given to M0 as the current address. At this point, one further instruction in the old instruction stream will be executed (the one that was fetched by M0 this cycle)- it will be distributed next cycle. After that, new instructions will continue to flow from M0.

We should note that this call invocation, although complicated, does not use the value stack, the ALU, or either of the data memories (M1 and M2). These functional units can be used to complete the argument list for the function being called. The current compiler and optimizer in fact *do* overlap the completion of the argument list with the invocation of the call, so for all function calls of more than zero literal following more than zero calculated arguments, there is no overhead whatsoever in calling a function except for fetching the arguments.

7.1.3.3. Return Stack Usage

Returning from a function invocation is done similarly to execution of a call. The data flow in this case is from the return stack saved address to the sequencer and M0, and from the return stack saved pointer to the value stack. The value stack is cycled once, to write the returned value into the proper location on the stack. This leaves the value stack available for all but the lastmost cycle in the return, and the ALU and both data memories available for both return cycles. Similarly to the procedure used in calls, the current compiler takes advantage of these available cycles to complete the computation of a result *after* the return instruction has been executed but is still in the program-visible pipeline.

The careful reader will note that this optimized calling system will fail miserably if one call instruction immediately follows another. Instead of having both subroutines executed in order, the first instruction of the first routine will be executed, followed immediately by the entire second routine. On return from the second routine, the remaining $n-1$ instructions of the first routine will be executed, and then control will return to the main program. The current compiler and optimizer are aware of this phenomenon and interlock against it, by including pseudo-operations in each such dangerous instruction such that two such instructions will never be optimized to less than two instructions apart. The actual cases where this occurs are rare, and when it occurs, the machine merely slows down a little. Currently, the only cases of this type are the exits from conditionals where no clause was satisfied.

A second type of frame is sometimes placed on the return stack. This kind of frame is called a "fake return frame" and its purpose is to store a branch location within the current frame, in a recursive fashion. This is necessary in the case of CONDs and other nested control structures. A fake return frame carries the address of the target code in its instruction slot, and a copy of the argument pointer in its argument pointer slot. Thus we gain the advantages of increased bandwidth in local branching as well as global function calling.

7.1.3.4. Binding Stack Usage

According to the syntax of Common LISP, arguments to a function are only visible within that function. To support globally-visible but dynamically scoped objects, Common LISP allows a dynamic binding to be established on an object via the `progv` special form. In other LISPs (ZetaLISP, MACLISP, etc) the default binding form is global visibility and dynamic scoping. To further complicate matters, catch frames are defined as dynamically scoped even in Common LISP.

In order to support this scoping quickly, the binding stack is used. When a symbol is dynamically bound, the symbol address and the old binding are pushed onto the binding stack. To unbind a symbol, the address and old value are popped from the stack and are written back into the symbol table. In essence, the binding stack and M1 provide hardware support for a shallow-binding variable scoping system.

The binding stack is accessed often during catch-throw activity. Because dynamic bindings must be unwound in the correct order, the binding stack is repetitively cycled during a `THROW` to do the unwinds. Because catches themselves are dynamically scoped, the catch frames are expressed as binding frames and the `throw` routine need only look at the tag bits of the address in order to decide whether any particular binding frame represents a binding to be undone or a catch which might receive the throw.

7.1.4. ALU

The ALU has two 256-bit inputs and two 256-bit outputs. It is assumed that the ALU can perform most functions in one nominal cycle (100 nanoseconds).

The ALU contains four important sub-modules, and several multiplexors to control the flow of data between submodules. All submodules are strictly combinational (memoryless) circuits. The four ALU submodules are a boolean module, a multi-purpose adder, a barrel shifter, and a status code generator. Only the multifunction adder and the status code generator require bit interaction across the 256-bit results.

Data flow within the ALU can be seen on figure 7.1.4a . Two inputs from the distribution bus system are latched upon entry to the ALU module on the rising edge of the processor GO signal. Command bits from the sequencer control the three internal multiplexors as well as the operation of the boolean module, the particular

function performed by the adder, and the amount of shift (or rotate) provided by the barrel shifter.

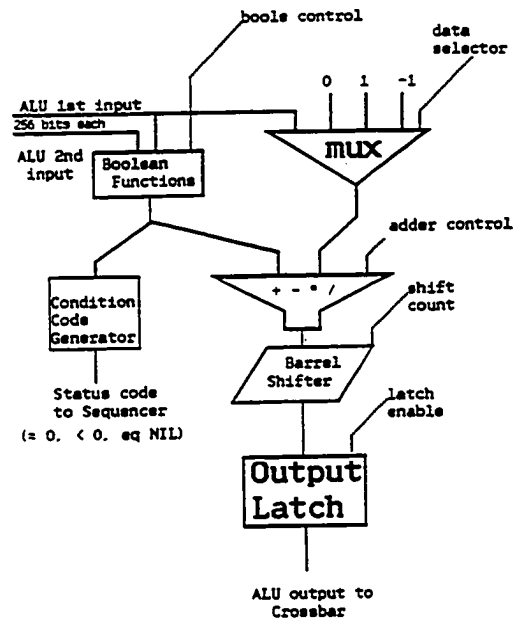


Figure 7.1.4a The DIS ALU Unit

Data coming from the distribution bus immediately go into the boolean operator submodule. This submodule allows all 16 possible boolean operations on two variables. This encoding requires four bits and produces a 256-bit result (BOO). Condition codes are generated directly from the output of the boolean box.

The multifunction adder takes the boolean output BOO as one operand, and has a multiplexor switch controlling the choice of four second operands. The operands available on the second input are the first ALU input, 0, +1, and -1.

The multifunction adder has two major modes of operation: fixed point and floating point. In an actual implementation, there would be separate hardware boxes with the output selected by a multiplexor. Conceptually all arithmetic operations are performed by this one submodule.

In fixed-point and floating-point modes, add, subtract, multiply and divide are supported. The result of floating-point operations are 80-bit IEEE format values, but the fixed-point versions are carried to a full 256 bits of accuracy. This facilitates rapid operation on bignum numbers.

The result of the multifunction adder is then supplied to a barrel shifter which

provides an arbitrary rotation, and then tag bits are masked in. The result is available on the ALU distribution bus AD0 on the cycle following the current instruction.

7.1.5. Sequencer

The sequencer module is responsible for next address calculations, for distribution of the command bits to the various modules, and for control of the MODULE-WAIT and GO synchronization lines.

The sequencer has three inputs from the distribution bus system. The first input is used to read the next instruction to be broadcast to the functional units, and therefore usually reads from M0. The second and third inputs are the primary and secondary next address inputs. The use of these two address inputs are discussed in detail below.

The sequencer has two "outputs", one being a distribution bus value, the other being the instruction being distributed to the various functional units. Only the output on the distribution bus can be used as a datum by another functional unit.

7.1.5.1 Next Address Processing

The sequencer generates two possible next addresses during each machine cycle. The generation path is identical for the primary and secondary address inputs. First, a small constant (the integers zero, one, two, or three) is added to each of the address inputs. Then, a multiplexor is used to choose between one of the two results. The result chosen is then output on the sequencer's distribution bus. Figure 7.1.5.1a shows the sequencer next address generation logic.

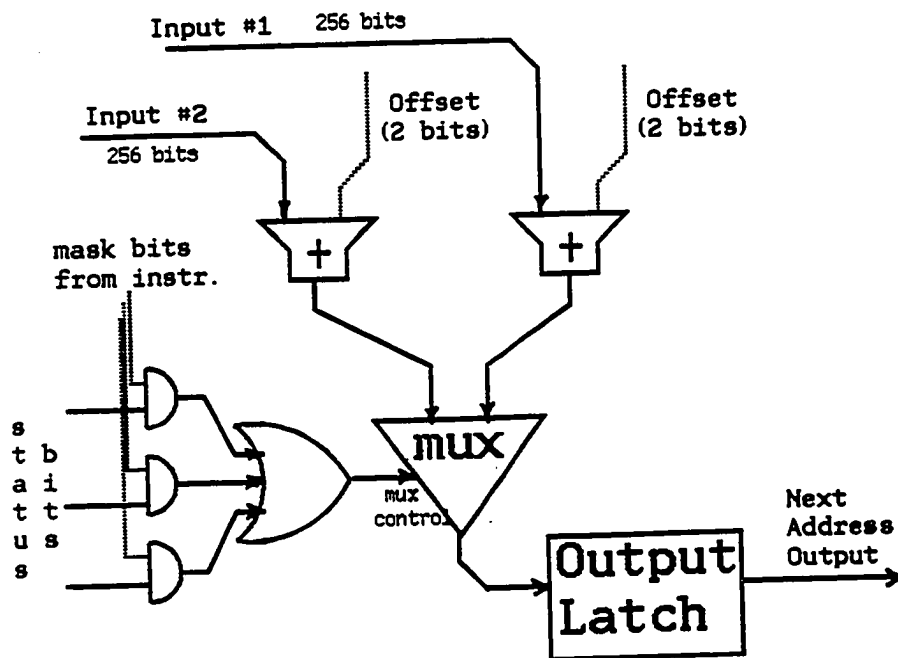


Figure 7.1.5.1a The DIS Sequencer Address Generator

The multiplexor control is composed of a number of status lines coming from various parts of the machine, a series of AND gates allowing each line to be masked or unmasked, and a multiple input OR gate. The multiplexor is thus defined to choose the primary input EXCEPT when an unmasked status line is high.

Status lines may come from anywhere, but most come from the ALU. Three lines are available, and reflect the value generated by the boolean operator box in the ALU. These lines are currently set on word equal to zero, numeric value less than zero, and pointer value eq (in the LISP sense) to NIL. These bits are available six "long" gate delays from the start of an instruction, so there is plenty of time to switch the sequencer output multiplexor during the instruction. R. Shane has proposed adding additional lines for use with extended data tags.

7.1.5.2. Command Word Distribution.

The sequencer loads a command word for distribution on the rising edge of GO. This word is held in a latch and driven out to all other modules on the rising edge of MODULE-WAIT. The command latch is read from the distribution bus system, and hence can use the output of any module as an instruction.

The command latch generally contains the output of the M0 memory unit. Because bit fields in the command field are preallocated at the module level, there

is no instruction decode delay; the same control bit is always routed to the same module. By having no instruction decode time, I feel a nominal cycle time of 100 nanoseconds is reasonable. If the cycle time were any shorter, (say, 10 nanoseconds), the memory and stack units would not have sufficient time to complete a read or write operation, and those units would be consistently causing the sequencer to issue waits via the MODULE-READY lines.

7.1.6. Network Interface

The Network Interface acts to field memory cycles which refers to memory outside of the local processor. In this sense, it (and the other network interfaces it is attached to) act as a memory server. When a request for a local memory read comes in over the net, the network interface acts as a memory requestor. The network normally references the M0, M1 and M2 memory partitions. In the most common case (a read to the M0 program area) there is no cache invalidate needed because the M0 unit is never used for a write operation.

In order to speed references the memory devices are given addresses in such a way that any reference to a possibly shared memory area (or to a memory area which is resident on another processor in a multiprocessor DIS machine) is a reference to a network interface partition. In this way the network hardware is automatically utilized to check for cross-network references, in whatever way is deemed appropriate.

Network communications for task sharing is performed by having the vacant processor actively check for a task to perform. The network interface provides three bits of information, encoding three states. The bits are available for testing in the condition code part of the sequencer.

The network task-related states are No Task Available, At Least One Task Available, and Too Many Tasks. If a task is available, the processor may "accept" it. If no task is available, "accepting" a task feeds the processor a dummy task which contains only a return instruction (followed by the no-ops necessary due to the prefetch- see under "sequencer"). Accepting a task simulates a subroutine prepare-to-call combined with overwriting the top two elements of the value stack (which are the arguments to the accepted task). Because of the overwrite, a processor executing a network task accept must first place two words of garbage on the value stack to reserve storage for the two values passed by the task accept.

Because testing for the presence of a task is independent of accepting it, the processor may also make a local decision based upon the current state of busyness of the local part of the network. For example, if the multiprocessor is full of tasks, it would be unwise for a processor to add another task. Instead, it should execute the function locally, trading time for a certainty of finding an available processor. Likewise, if there are few tasks in the system, a processor may split the current task, if the task is applicative and there are processors available.

Submitting a task requires a pop on the pending stack and a copying of all value stack elements between the value stack pointer and the pending subhead pointer to

the network. For all program purposes, the call "disappears" from the local processor and "appears" in another processor somewhere in the network. Because execution of the submit removes the call from the stack completely, submit and accept provide a fast process dispatch mechanism, with low cost to interlock and an easy load-management mechanism.

7.1.7. Booting, Debugging, and The Host

On power-up, the memory and hardware registers of the processor will no doubt contain garbage. Before processing may begin, the proper state and initial program must be loaded into the processor hardware. This is accomplished by either a host processor or a DIS processor-local Boot-Debug Processor (BDP). BDP's are only needed in a multiprocessor environment.

We envision that the host processor will not only boot the DIS machine, loading a saved system into the memories and initializing the stack pointers, but will also do all physical I/O for the DIS machine. This will probably be implemented by a 32-bit parallel interface between the host and the DIS machine, as well as a number of DIS status and control lines being available by the host. Because there is no provision in a DIS machine for interrupts, the host processor will be responsible for fielding all interrupts as well as handling all user interaction. In this way, the host or BDP frees the DIS machine to do what DIS machines do best; run LISP.

The BDP is a conventional processor such as a 68020 or a MicroVAX running a server program and with a small amount of local memory. The BDP has access to the processor memory as well as to the hardware access "partition", called the boot/debug partition. The boot/debug partition of memory contains the MicroVAX local RAM and also accesses all the internal registers (such as stack pointers and display registers) of the local processor. This allows the BDP to load an initial state into the processor and to single-step the processor (by forcing a low state on MODULE-WAIT). Each BDP also has a unique serial number, which is used to address a specific BDP and also gives the processor a unique number.

BDPs in a multiprocessor system do not communicate via the Network Interface. Instead, the BDP's share a common 8-bit bus and have a daisy-chained attention line to the console BDP. The BDP's may receive broadcast messages from the console, or a single BDP may be selected by the console for some detailed operation. During a cold boot, the console BDP broadcasts the initial program to all processor BDP's. Should any module (such as the network interface module) become sufficiently complex that it requires microcode, then the microcode area should also be accessible via the boot/debug partition of memory, and can be loaded by the BDP during cold boot.

7.2. Intraprocessor Interconnection and Parallelism

The various modules of a processor communicate via sixteen distribution busses. Each module with a globally accessible output drives a distribution bus. Each module with a distribution system input may read any of the distribution busses.

For example, memory M0 has one 256-bit output, which is available to all other modules via the B-MD0 distribution bus. Likewise M1 has a 256-distribution bus, B-MD1.

Previous ideas for the DIS architecture had certain hardwired assignments of interconnection. In the process of designing the compiler, it was found that such hardwired interconnections were almost invariably accelerated one particular circumstance (such as function calls with one argument) and impeded most other circumstances. The regularization of interconnection via the distribution bus system is as fast as the previous designs, and has much greater flexibility, which the current compiler does use to increase parallelism.

The full list of distribution busses is as follows:

B-MD0	M0 data output
B-MD1	M1 data output
B-MD2	M2 data output
B-VDO	Value Stack Top Datum
B-VD1	Value Stack Below-Top Datum
B-VSP	Value Stack Stackpointer
B-PDO	Pending Stack Top Datum
B-PD1	Pending Stack Below-Top Datum
B-RDO	Return Stack Top Datum
B-RD1	Return Stack Below-Top Datum
B-BDO	Binding Stack Top Datum
B-BD1	Binding Stack Below-Top Datum
B-ADO	ALU output
B-SDO	Sequencer Next Address
B-NDO	Network Interface Datum
B-CMI	Command-Immediate Datum

Each distribution bus may be read by any or all of a number of input latches. The distribution bus system may be viewed as a crossbar in this sense.

The full list of input latches is:

MOD0	M0 address latch
MOD1	M0 data latch
M1DO	M1 address latch
M1D1	M1 data latch
M2DO	M2 address latch
M2D1	M2 data latch
VVO	Value Stack Top Datum

VV1	Value Stack Below-Top Datum
PV0	Pending Stack Top Datum
PV1	Pending Stack Below-Top Datum
RV0	Return Stack Top Datum
RV1	Return Stack Below-Top Datum
BV0	Binding Stack Top Datum
BV1	Binding Stack Below-Top Datum
AV0	ALU first input
AV1	ALU second input
NV0	Network interface datum
SV0	Sequencer Next Instruction
SV1	Sequencer Primary Address
SV2	Sequencer Secondary Address

7.2.1 Construction of the Crossbar

Actual construction of a DIS crossbar is nontrivial, but it should not be impossible. To assure ourselves of this, let us consider a few possible methods of implementation.

7.2.1.1. Option 1: Brute Force Crossbar

We could attempt to construct a DIS crossbar in the obvious way; by buying a large number of cables and multiplexors. Let us examine the results of this method.

Each DIS distribution bus is 256 bits wide. There are 16 such busses, each bus feeding 20 multiplexor inputs. Each multiplexor would drive 256 bits of output to a functional unit output. The total number of signal pins on a hypothetical crossbar would then be $(16 + 20) \times 256$ or 9,216 pins.

This is clearly a hard thing to build, so let us partition the task. First, we observe that a fan-out of twenty is on the edge of acceptability without recourse to special driver chips. Therefore, we accept production of sixteen distribution bus driver cards, each one producing sufficient drive to control twenty inputs, over a 256-bit bus. This requires 512 pins, which is acceptable. We then need to construct twenty switching devices, each one reading all 16 distribution busses, and selecting one for input to a functional unit.

These sixteen-bus switches require (16×256) or 4096 inputs, plus 256 bits of output, total 4352 pins. We can partition this module into 64-bit slices, giving 1088 pins per card. Of these pins, 1024 are inputs and 64 are outputs. We can pack the 1024 input pins in groups of 64 via standard connectors, giving 16 such connectors per board. Each connector will take up about 3 square inches of board area (standard Eurocard connector), so about 48 square inches or .34 square feet of area per board will be dedicated to the input connectors. Another three square inches will be used for the output connectors. The 64 multiplexor chips needed will take on the order of .5 square inches each, totalling 32 square inches. Total size of a switch card is then

about 83 square inches, slightly smaller than a standard 8 1/2 by 11 sheet of paper. We will need (20 x 4) or 80 such cards, all identical.

A VAX-11/780 has approximately sixty cards, each card being about twice the size of the above-postulated switch or driver card. This gives about one hundred and twenty switch-card equivalents per VAX, which leaves us with 24 switch-card equivalents if we wish to maintain the VAX form factor. This is a reasonable size for a prototype machine and hence we might conclude that even with brute-force design, the DIS crossbar is quite buildable.

7.2.1.2. Option 2: Microwave or Fiber Optic

optic If we choose not to build a wide crossbar, we can still construct a crossbar with full capabilities if we use a sufficiently fast technology. In order to transfer a full 256 bits of data in 20 nanoseconds (assuming Manchester-coded signals for easy modulation/demodulation) on a bit-serial line, we need a bus bandwidth of 25 GHz. This is K-band microwave, and in fact, it is incredibly convenient. The common use of police speed radar (and hence, police radar *detectors*) in our society has pushed production of police K-band (25.525 GHz) devices into the millions per year. Cost per device is on the order of one to ten dollars retail.

We will now contemplate a DIS crossbar system composed of sixteen K-band waveguides (approximately 1 cm by .4 cm), each drilled in eleven locations to accept an insulated probe. Each DIS functional unit card has sixteen insulated probes to tap the microwave signal out from each waveguide, plus an additional "drive" probe to transmit an output signal onto each dedicated waveguide/bus. Each DIS card, when mated into the microwave backplane, can then select whichever waveguide it wishes to read in order to acquire input data.

In this configuration, we need 320 microwave transistors to act as pass switches, twenty local-oscillator diode/detector diode pairs to demodulate the microwave signal and twenty 12.75 GHz shift registers to capture the incoming data stream. This is clearly constructable if we use GaAs shift registers.

If we wish to use laser diodes and fiber optics instead of microwave interconnects, we need 320 laser/photodetector pairs, and twenty shift registers. Precision construction techniques for drilling and aligning the backplane are not necessary with the flexible fiber optics.

The advantages of the microwave/fiber optic system over the brute-force crossbar is in size. The fiber optic crossbar should take on the order of 40 square inches per functional unit, so we probably do not need any additional driver or switch cards as we needed in the brute-force crossbar configuration. This shrinks the minimum size for a DIS machine from one hundred cards to ten cards.

7.2.1.3. Option 3: Restricted Configuration Crossbar

If we examine the actual usage of bit fields in code produced by the current compiler and optimizer, we find that the majority of the high-order bits on M1, M2, and the stack units are unused. If we modify the current compiler, we can produce code which uses only the low-order 32 bits per word on all units *except* M0, the Sequencer command input, and the distribution bus connecting those two devices. This interconnect is shown in figure 7.2.1.3a .

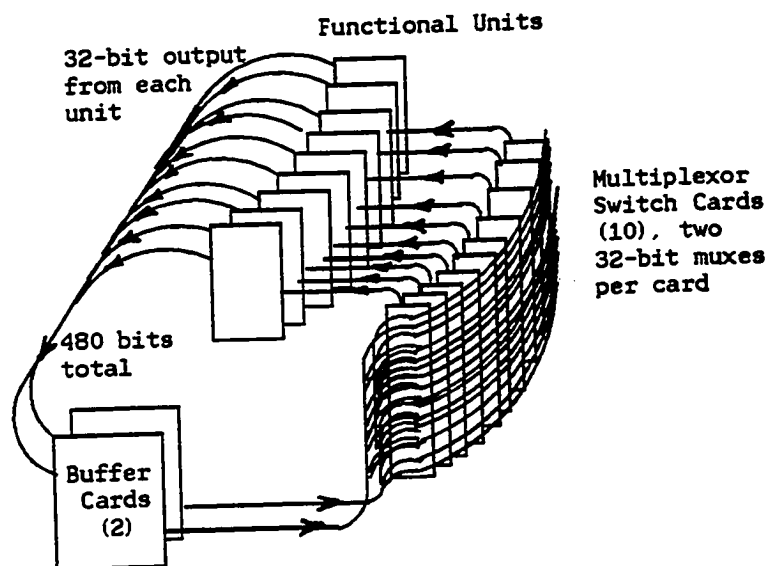


Figure 7.2.1.3a Restricted Crossbar

Let us examine this in more detail. If we interconnect M0 and the sequencer command input directly (remove it from the crossbar entirely), we need construct a crossbar of size 15 x 19, with a width per bus of only 32 bits. Using logic similar to that used to determine the size of the brute-force crossbar, we find that we need two drivers card instead of twenty, and ten switch cards instead of eighty, each switch card being split into a pair of 32-bit switch paths instead of a single 64-bit switch path. One each driver path and switch path remain unused in this configuration.

By going to a restricted crossbar configuration, we can shrink the size of a minimal DIS machine from one hundred cards down to twenty-two notebook-sized cards, with only slight loss of functionality. The BSP [Kuck 1982] used pair of "alignment networks", each of which was similar in size and complexity to the proposed DIS crossbar.

●

If M0 can be directed to begin fetching the new instruction stream with certainty (an unconditional branch or call), one instruction past the current instruction will be executed before the new stream begins execution. If M0 must look at the output of the sequencer, then two instructions past the current instruction will be executed before the new stream begins execution. The current compiler knows about unconditional branches and will give M0 the needed address early in order to minimize this branch latency whenever possible.

To help visualize this user-visible pipeline, consider the following example (with accompanying figure 7.3a)

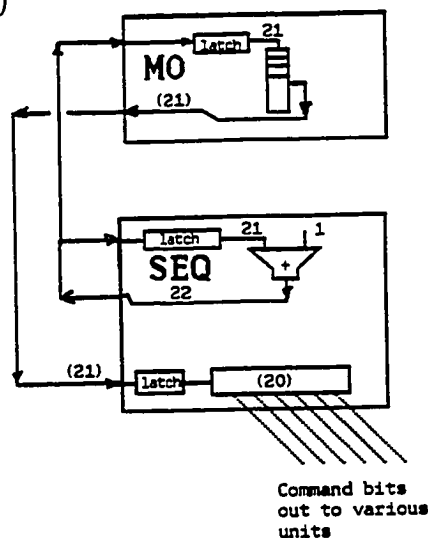


Fig 7.3a The DIS User-Visible Pipeline

In this example, (seen near the end of the cycle), the sequencer has just generated absolute address 22. M0 latched it's data at the beginning of the current cycle, so it is currently completing the fetch of location 21. The sequencer command output latch is currently driving the instruction fetched last cycle (address 20) to the various functional units.

7.4. Garbage Collection in DIS

The DIS architecture does not have special-purpose hardware for garbage collection. It was found in the design stage that the complexity of hardware required for automatic garbage collection was actually greater than the complexity of all other parts of the DIS architecture together. Instead, the DIS machine does garbage collection in software.

Garbage collection in DIS is triggered by a storage-allocation function attempting to allocate a cons cell or array, and finding none available. The storage-allocation function then calls the garbage-collection function, which traverses all data structures, marking them as to accessibility, and then creates a new free list from the inaccessible cells. Garbage collection uses the ALU's ability to perform boolean operations and observe the result in a single cycle, rather than by using microcode-controlling tag bits.

The DIS architecture does not check tag bits directly (for garbage collection or any other purpose, excepting the reserved bit pattern for NIL). Instead, software checks the tag bits by ANDing an immediate constant in the ALU with the word in question, and then performing an XOR with a word of the type desired. If the status code is zero, the tag bits match, and the questioned word is of the same type as the desired type word.

Garbage collection does not need to transverse all memories and stacks in a DIS machine. M0 never contains pointers, hence, it is never visited. M1 contains the symbol table (which is of known length), and so must be visited, but it is unnecessary to perform any markings in the symbol table. M2 contains cons cells, and must be both traversed and marked for collection. Only the Value and Binding stacks of a DIS machine contain pointers to data structure, so only those units need be accessed during a garbage collection. Both stacks contain only pointers, so we need to perform only traversal and not marking on the stack units.

It must be stated that although we have worked out *how* garbage collection works in the DIS architecture, we have not actually implemented it. The current simulator is far too slow to test a garbage-collection implementation.

7.4.1. Estimate of Time to Garbage Collect

If we choose some reasonable numbers for the DIS machine and for the program environment, we can estimate how long the DIS machine takes to run a complete garbage collection.

Let us assume there is two megawords of consing space total, and one half of the space is actually in use (the other half being garbage). Also, let us assume there are ten thousand active symbols in the symbol table, and five thousand procedures. Similarly, we assume that a thousand objects are dynamically bound, that the current recursion depth is one thousand, and that each recursion level has ten

intermediate pointers. These assumptions are probably overly generous, but since we wish to look at a worst-case scenario, this is reasonable.

We define a measure of action called a "visit". A visit is the action taken when a single pointer is followed to what it points to, and the pointer itself marked with a tag to show that it has been followed. First we will count visits needed to mark all active cells, and then we will count visits needed to collect all the inactive cells into a new free list (called "mark and sweep" garbage collection).

We know that the ten thousand symbol table objects each have at most four pieces of list structure associated with them, that being the symbol-value, symbol-function-text, property-list, and object-list. At most, this requires forty thousand visits, each visit marking one cell in the M2 memory space. Each of the thousand objects dynamically bound must also be marked into M2 space, and accepting that each object on the value stack is also a pointer (a very safe assumption), this adds another one thousand and ten thousand visits, respectively.

Now the only cells which must be visited and followed are in M2. There are (by our assumptions above) one million such pointers; on the average each one will be pointed to by just over one other pointer (each full circular list structure will add .000001 to this average). To visit each of these pointers will require on the order of one million visits.

We see now that the fifty-one thousand visits required to mark all the cells pointed to by M1 (the symbol table), the Value Stack, and the Binding Stack are completely dominated by the million visits needed to follow the pointers pointing back into M2. In our scenario, to mark every accessible cell requires on the order of a million visits. The reader should note that we do *not* do all the M1 marking and then Value Stack marking, etc. Rather, each tree of list structure is followed to completion before the next symbol table entry or stack entry is begun. Although the order is different, the number of visits is the same and only the number of visits makes any difference.

How long does one million visits take? Because M2 is only single-ported, we will assume that no pipelining can take place. We need one cycle to read the pointer, four to check its tag bits (and return if the cell has already been visited), four to check if the pointer points to NIL (but this can be done overlapped with the previous check, so only one extra cycle is needed), one to set the tag bit, and one to write the word out. Because these are cons cells rather than single pointers, we must maintain a reverse path to follow the second pointer of the cons cell (either by stacking the pointer location on some stack unit or by reversing the pointer chain; the pointer-chain reversal method is described in [Hillis 1985]). Either method takes an extra two cycles (either to stack or memory).

The basic concept of pointer-chain reversal is that the car path is followed to the end, but in each car operation, we replace the car with a pointer to the cell that pointed to it (with the "visit" tag bit set). Now the tree has been reversed, and we can move back up the tree, un-reversing the pointers and following the cdr part of

the tree recursively as we go. The result is that two extra memory cycles (an extra write, then an extra read) must be run, but that we do not need any space on the stack to run this marking phase of the garbage collection.

Once all accessible cells have been visited (at a cost of ten million cycles), we must sweep through M2, chaining all the unmarked cells into a new free list and resetting all of the accessibility tags. This requires a second visit to each cell. We need to read the cell, check the tag (four cycles), clear the tag, and write the cell. For a marked-accessible cell, this is seven cycles. If the cell is not marked accessible, we need to write the previous end-of-free-list chain (one cycle), and update the free-list-end (two cycles). Thus, eight cycles are needed for every garbage cell, and seven for every in-use cell (a cell contains two pointers, only the first of which will be marked). This totals to four million cycles for garbage cells, and three and a half million cycles for in-use cells.

We can therefore conclude that a complete garbage collection in a two-megaword DIS machine will take on the order 1.8 seconds (eighteen million cycles). Best case will be with almost no in-use cells, in which case eight million cycles will run (.8 seconds), and worst case will be when almost all of memory is useful data (2.7 seconds). This is probably too slow for real-time system use.

It might be possible to implement a continuous collection system (every call to cons runs three cycles of mark, or three cycles of sweep), but we have not explored the overhead inherent in such a scheme.

7.5. Concluding Design Remarks

This concludes the abstract machine definition of the DIS architecture. There are still a few rough spots concerned with the details of the network interface. That part of the design is still evolving, and a complete description should be found in R. Shane's thesis (pending publication).

The user interested in assembly-level programming of the DIS architecture is now referred to appendix B, "The DIS Instruction Summary", and to chapter 9, "Examples of DIS Machine Code". The user is also hereby warned that the DIS LISP compiler and optimizer can probably produce code at least as efficient as the naive DIS assembly-language programmer, and that the compiler is aware of the user visible pipeline of the DIS architecture and will avoid pipeline hazards automatically.

The DIS architecture provides an environment rich in potential parallel paths. The next question is: Can we automatically utilize the potential parallel paths in a DIS machine and the potential parallelism within LISP expressions in order to gain a large speedup in LISP execution speed? The answer to that question appears to be an unequivocal yes, which is demonstrated in the next two chapters.

Chapter 8

The Compiler, Optimizer, and Simulator

This section describes the software used to evaluate the performance of the DIS architecture.

The DIS simulation system is written entirely in VAX Common LISP, from the compiler through to the execution simulator. Overall, there are over 5000 lines of LISP code. When loaded into the VAX Common LISP environment, slightly over 450 Kbytes of space is needed hold the entire DIS simulation system (source code only).

The DIS simulation system has four major parts: the compiler, the optimizer, the linking assembler/loader, and the simulator. Although each part can be considered separately, the reader should realize that a number of macros define pieces of the DIS system for use in multiple places.

The DIS design uses these system-definition macros in order to assure that the compiler, optimizer, and simulator remain consistent with each other. Because up to four people at once have been actively developing the DIS simulator, compiler, and optimizer, some means had to be established to assure that bit and field allocations were consistent throughout the design.

Figure 8.0a gives an overview of how the DIS simulation software interrelates.

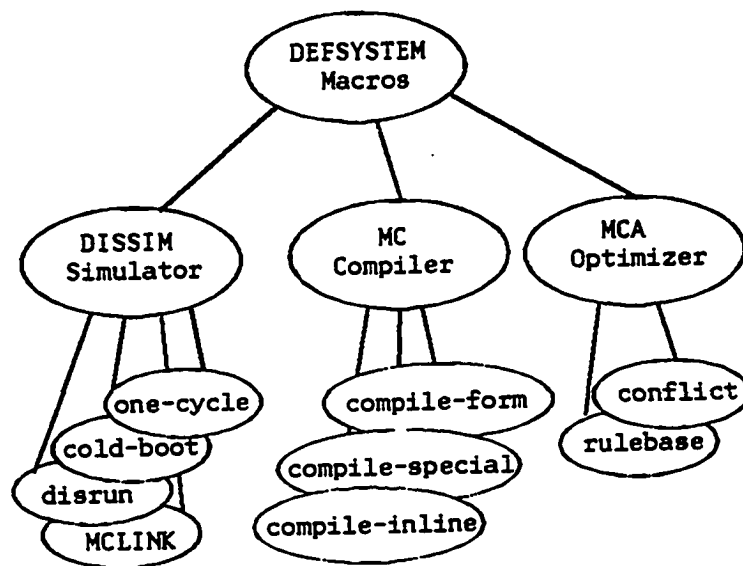


Figure 8.0a DIS Simulation Overview

8.1. System Definition Macros

The `deffield` macro is an example of a system-definition macro. `Deffield` defines a bit field in an instruction word, creating a symbol of the name `f-<fieldname>`. `Deffield` also creates a selector function `s-<fieldname>`, which extracts the given field from an instruction word, and another function `<fieldname>` which places a numeric argument into the field specified. In this way, `deffield` and other macros are used to not only define a computer architecture, but to simultaneously create that architecture's simulator and generate an assembler for that architecture.

Macros of this type are `deffield`, `defop`, `defbus`, and `defspace`. These macros are defined and used in the file `disdef` (listed in appendix 1). The reader is referred to that file for greater detail of these macros.

The DIS simulation system is designed to be used similarly to the `TASKBUILDER` of `RSX-11`. Modules are compiled, and as they are compiled, a partial symbol table is constructed. When the modules are loaded, addressing references in the machine code are resolved by two passes of the linking loader. Because the compiler only knows that a function `foo` exists, but not where, two passes of the linker are required to resolve all function location information and produce both a complete and updated DIS symbol table in `M1`, and an executable code image in `M0`.

8.2. Compiler Internals

The DIS compiler is a recursive-descent compiler which generates a vertically-oriented microcode stream directly from a LISP expression. The current compiler, called `MC` (for Microcode Compiler) generates assembler instructions and assembler directives for single-processor programming.

Because the LISP `read` function creates tree-structured lists directly (in fact, the LISP user must take great pains to read in a list structure as a non-tree datum), there is no parser or tree generator in `MC`. Instead, functions are compiled into vertical microcode from LISP S-expressions already read into memory by the Common LISP interpreter.

`MC` generates both symbolic machine instructions and directives to the optimizer and the loader. Because of the highly parallel nature of the DIS architecture, an instruction often contains caveats such as "no other use of the value stack is permitted in this instruction", or "this instruction must remain within 2 cycles of the end of this basic block".

8.2.1. Recursive Descent Modules

The top level of the compiler is the function `MC`. `MC` uses the dynamically scoped variable `result` to provide a place for emitted code to accumulate. `MC` then passes the form to be compiled to `compile-form`.

`Compile-form` examines the form and, depending on the form, passes the form on to one of `compile-a-special-form`, `compile-an-inline-form`, `compile-a-non-special-form`, or `compile-value-reference`. Which of these is called is determined by the data lists `*special-forms*` and `*inline-forms*`. The compiler assumes that any form not on one of these two lists is neither a special nor an inline form and compiles a reference to the `symbol-function` or `symbol-value` cell of the symbol table to get the appropriate information.

The special forms currently supported are `defun`, `cond`, `setq`, and `let`. The file `mcspecials` contains LISP routines to compile each of these special forms. These forms are adequate to run all the benchmarks listed.

Because each special form may itself contain nonspecial forms or implied `progn`'s, the compiler may recurse back to `compile-form` or may call `compile-implied-progn` to produce the necessary code.

Of special interest is how the `cond` special form is compiled. Because a `cond` may exist in a recursive loop, it is not feasible to have a statically determined exit location for a `cond`. Instead, a "fake return frame" is generated and pushed onto the return stack. This frame contains an unchanged value stack pointer, but the return "location" is really the location of the end of the `cond`. In this way, a clause within a `cond` can jump past all remaining clauses in a simple and clean way, by executing a `Return`. In most cases the optimizer will overlap the firing off of the `Return` with the last part of the clause evaluation.

The `cond` special form is "smart" about the value of `T`, in that a clause whose predicate is `T` does not evaluate `T`, it just assumes that `T` hasn't been rebound to `NIL` by the user and evaluates the consequent clauses.

Inline forms are compiled by functions in the file `mcinline`. The currently supported inline forms are `+`, `-`, `*`, `/`, `car`, `cdr`, `1+`, `1-`, `eq`, `rplaca` and `rplacd`. These forms generate short microinstruction sequences which accept their arguments directly from the value stack and return results directly on the value stack. These routines could have been written as a system software library, but the microinstruction sequences involved are so short that it did not seem worthwhile compared to the increased possibilities for action by the optimizer given by inline code.

8.2.2. Symbol Tables

During compilation of a function, two symbol tables are accessed. The first symbol table, named `symbol-table`, "grows" with each compilation, and is a representation of the DIS machine symbol table held in memory M1. The second symbol table is the lambda-variable symbol table `locals`, and is dynamically scoped within the `compile-a-defun` recursive descent form.

Symbols are added to the global table `symbol-table` whenever the first reference to them is seen by the compiler. As in other Common LISP interpretations, a variable not declared lexical (by appearing in an argument list) is assumed by the compiler to be global and dynamically scoped (a "special" variable according to Steele).

An entry in the M1 symbol table is eight elements long. This implies that a pointer to a symbol is always a multiple of eight. Because address space is not a problem in the DIS architecture, we directly use this multiple of eight as a sequence number without an offset, giving an address into M1. If M1 is defined as memory starting at location 1000, the first symbol (defined as T) will have sequence number 1000, NIL will be at 1008, and the pointer to free cells (for consing) will be at 1016.

The format of the M1 symbol table is as follows:

Displacement:	Contains:
0	symbol-value-pointer
1	symbol-function-pointer
2	symbol-name-pointer (printname)
3	symbol-plist-pointer (property list)
4	symbol-package-pointer
5 - 7	reserved for a flavor system and other expansions

Because the current compiler does not support packages, and property lists are not modifiable at compile time, a `symbol-table` slot is only four elements long. Symbol-table entries are of the form (symbol-name symbol-table-slot-address symbol-value-ptr symbol-function-ptr) .

If it is necessary to upgrade the compiler to have a more detailed symbol table, only the functions `make-symbol-table-entry`, `symbol-seq-lookup`, and `mc-hard-init` will need to be altered.

8.2.3. Code Generator Stubs

Each of the compiler routines uses a set of lower-level service routines to actually emit code onto the `result` special variable. These routines are the `app-res-xxx` routines in the file `mcstubs`. These routines generate one- to four-instruction sequences which perform tasks at the lowest level in the DIS architecture. These routines "know" how to emit code to perform their designated tasks, along with the extra information concerning forbidden interactions that is needed by the optimizer.

These code-generating stubs are:

```
vpush-from-cmi
prepare-to-call-vs-addr
prepare-to-call-symbol
execute-prepared-call
symbol-function-location
anonymous-location
pop-vs-one
pop-rs-two
return-to-caller
return-via-fake-frame
vpush-from-local-values
set-local-variable
set-symbol-value
symbol-value-lookup
symbol-function-lookup
create-fake-return-frame
bs-push-global-symbol
global-unwind-bs-and-pop-two
branch-to-location
branch-if-null
```

Additional service routines emit code for the following inline functions:

```
add-vs-pair
subtract-vs-pair
multiply-vs-pair
divide-vs-pair
pop-vs-1-overwrite-ALU
add-vs-one
subtract-vs-one
vs-car
vs-cdr
vs-eq
vs-rplaca
vs-rplacd
```

These routines in general append instructions to the end of result. They may probe the symbol table (via symbol-seq-lookup) in order to encode an absolute address.

8.3. Optimizer Internals

The DIS MCA assembly language optimizer is a single-pass recursive-descent optimizer which uses compiler-generated constraints, sequence-checking, resource-checking, and a rule-based peephole system to produce a mostly-horizontal microcode from the vertical microcode generated by MC. MCA is able to produce about

a 40 % improvement over the code generated in MC by using knowledge about the intraprocessor parallelism available in the DIS architecture.

Considering that MC output code already contains significant parallelism (all of the low-level code-generating stubs were hand-coded for maximum parallelism; eleven of twenty access multiple functional units in a single instruction), an improvement of 40 % is significant. When compiling reasonably-sized programs, it is uncommon to see an instruction which does not activate at least four functional units (sequencer, M0, plus two others). Of the remaining functional units, contention is highest for the value stack, followed by the return stack.

From a heuristic point of view, the optimizer is a constraint-based system, with a rule-based system allowing overriding of the general constraints in certain specialized cases. The general constraints are similar to the constraints of a dataflow machine (waiting for data to become ready, and waiting for data to be accepted by the next unit in line), while the rule-based system allows special cases (such as multiple stack-push operations) to be combined. In this way, constraints which are too severe can be relaxed for those cases in which it is appropriate.

The three major parts of the optimizer are the basic block partitioner, the clash determination section (constraint-based system), and the resource merge system (rule-based exceptions to the constraint-based system)

8.3.1. Basic Block Partitioning

The first operation that MCA performs is basic block partitioning. A basic block is a section of code which has only one entry and one exit, and contains no branch instructions. Clearly, instructions within a basic block may be rearranged or combined in any order that does not violate data-flow or resource requirement constraints.

The function `make-code-segments` in the file `mccrush` achieves the breaking of a code stream into basic block segments. Each basic block is then mapped to an equivalent but faster basic block via `crush-a-segment`. `Crush-a-segment` uses the Common LISP `reduce` function to repeatedly attempt to crush a single instruction onto the end of a partial basic block. The resulting list of basic blocks is then re-chained back into a long code stream.

The function `crush-instruction` actually performs the crushing. `Crush-instruction` is given two arguments- the current partial basic block, and the new instruction. Clash determination is done on the last instruction of the current basic block versus the new instruction. Depending on the results of the clash determination, the new instruction might be appended to the basic block, merged with the last instruction of the basic block, or (if no conflict is found) `crush-instruction` may recursively call itself on the current basic block with the last instruction deleted. In this manner each instruction in a basic block is placed in the instruction stream at the earliest point where it can be executed correctly.

8.3.2. Clash Determination

Instructions are crushed onto the tail end of a partial basic block by determination of conflict, via `instr-clash-p`. The last instruction in the partial basic block is compared with `instr-clash-p` and a decision on how to optimize is made on that basis. Possible results of `instr-clash-p` are T (the instructions clash irreconcilably), NIL (the instructions may be reordered or combined freely), or a new instruction which is a legitimate combination of the two instructions.

Five kinds of conflict between instructions are recognized by the optimizer. These conflicts are resource conflicts, sequence conflicts, dataflow conflicts, reservation conflicts, and branch conflicts. Resource, sequence, dataflow, and reservation conflicts are detected by `instr-clash-p`. Branch conflicts are determinable without regard to the current basic block environment, and are handled in `crush-instruction`.

Some types of conflict, such as sequence or resource conflict, force an irreconcilable clash. Other types of conflict such as dataflow or reservation conflict, allow the two instructions to be merged, but not slid past each other.

A sequence conflict occurs when the second instruction needs the result of a functional unit which is used by the first instruction. Since the result is not yet ready, the second instruction must be delayed till the first instruction completes. Hence, sequence conflicts force an irreconcilable clash.

A dataflow conflict is the reverse of a sequence conflict. A dataflow conflict is triggered by the second instruction commanding a functional unit whose output is used in the first instruction. Since the crossbar latches will hold the functional unit output until the end of the cycle, a dataflow conflict allows the two instructions to be merged into a single instruction, but does not allow the second instruction to be moved ahead of the first instruction.

A resource conflict occurs when two instructions both wish to use the same functional unit at the same time. For example, if two instructions both require the use of the CMI (immediate data bus), a resource conflict exists.

In some cases, a resource conflict can be resolved by the rule-based merger. This is because some functional units have commands which have the same final effect as pairs of other commands. For example, pushing one value, then pushing another value, has the same effect as pushing two values. Please see the section on the rule-based merger for further details.

A reservation conflict occurs when the second instruction requires a functional unit be reserved for a following instruction, and that functional unit is already in use. Reservation conflicts allow the two instructions to be merged, but inhibit the movement of the second instruction before the first instruction.

A branch conflict is triggered when a `CRUSHMAX n` compiler directive is encountered by the crusher. These directives are a necessary consequence of the delayed

effect of branches on instruction execution. Without artificially limiting the amount a control-altering instruction may be pushed forward in a basic block, the optimizer described will generate incorrect code because the control-flow altering instruction will not clash with any other instruction in the basic block (since basic blocks are delimited by control-flow instructions), and will float to become the first instruction in the basic block.

8.3.3. Rule-Based Merger

The rule-based merger is defined in the file `mcpeephole` (appendix 1), and contains both the driving function `resource-peephole-fixup` and the rule list `*resource-peephole-options*`.

`Resource-peephole-fixup` accepts a pair of instructions which otherwise would have irreconcilably clashed, and attempts to match the pair with one or more rules stored in `*resource-peephole-options*`. More than one rule can be applied to a pair of instructions. Any rule which applies will be executed, and the remaining rules in the sequence will be applied to the result of the preceding rules.

The format of the rules for the resource merger is a list of lists. Each sublist has four elements. Elements one and two are enumerations of opcode fields which **MUST** be found in the first and second instructions, respectively. If both the opcode fields and values are not matched in the proper instructions, the rule does not fire.

Element three of a rule element is "user data", used to execute the consequent of the rule. Element four is a lambda-definition to be executed to produce a new pair of instructions. The lambda definition is executed with an environment containing three arguments. These arguments are the first instruction, with the matching trigger opcodes removed, the second instruction with the matching trigger opcodes removed, and the user data (the third element of the list). The rule is expected to return a new instruction pair.

Note that because each rule is allowed to fire in sequence, and the result of one rule firing becomes the new argument for all subsequent rules, more than one rule may alter an instruction pair. A rule can be set to fire always by having elements 1 and 2 of the rule be null. Because the rule-based merger is not called on every pair of instructions, but only in cases of instruction clash, it isn't possible to force every pair of instructions to undergo some transform in the rule-based merger.

8.4. Linking Assembler/Loader

The linker `mclink` is defined in the file `mclink` (Appendix 1). The linker is a two-pass system which accepts as input an instruction stream and a user-supplied starting address. `Mclink` loads the memory of the simulator with an image of the executable and a proper symbol table, and returns the address of next available location.

On the first pass, `mclink` scans the instruction stream for `LOCATION` pseudo-opcodes. Each location name is queried against the compiler-generated symbol table in order to generate a location for the symbol-function cell. Both the M1 (hardware) location and symbol-table (software) are updated.

On the second pass, the individual instructions are assembled since pointers into the symbol table are now known. Because the compiler codes symbol-table locations directly into immediate data fields, it is necessary to have knowledge of all branch addresses before instruction assembly into bit fields can be completed. It would be possible to preassemble parts of the instructions but the DIS simulation system does it at load time. There is no loss of generality by doing this.

8.5. Simulator Internals

The DIS simulator per se (not including macros) is defined in the files `dissim`, `disgraph`, and `disalias`, with some basic functions in `franz2common` and `disbase` (Appendix 1).

As discussed previously, much of the simulator is defined by macro. The `deffield` macro defines a field in a horizontal instruction word, plus creator, and accessor functions for that field. Successive `deffield`s produce non-overlapping fields automatically.

The `defop` macro defines a valid symbolic opcode for insertion into a field created by `deffield`. Successive symbolic opcodes are created automatically. If too many opcodes are specified for a given width of field, an error is automatically issued.

The great advantage of this method of architecture specification is that the symbolic values such as `SPUSH2`, not hardcoded bit patterns such as "1001", are used in the functional unit descriptions.

8.5.1. Order of Execution

Because the DIS machine is a synchronous and parallel machine, some changes were made to simulate it on a conventional serial architecture. Execution of an instruction on a real DIS machine is of the following form:

```
All units latch data busses.  
All units execute.  
All units drive new results out.
```

In the DIS simulator, latching of all data is followed by cycling of memories in ascending order, then value, pending, return, and binding in that order, then the ALU, then the network interface, then finally the sequencer. As each functional unit is simulated, it updates its distribution busses (which are not seen by any other unit, because all inputs have already been latched).

The reader is cautioned to examine the "kanban" ALU condition code software carefully. Because condition codes are seen by the sequencer on the current cycle (not delayed one cycle), the condition codes are allowed to only express boolean functions of the ALU inputs, with a maximum nesting depth of two. Currently supported condition codes are "all bits zero", "high-order bit one (indicates a negative number)", and "EQ to hardware NIL". In this way, the condition code is available to control the sequencer's final multiplexor long before the two adders have completed producing the possible next addresses.

8.5.2. Functional Unit Simulations

Functional units of the DIS machine are defined by macros. One macro creates a memory unit, another creates a stack unit. Other macros define regions of memory address space. Currently, each memory unit is 1000 addresses long, and each Stack type memory is 2000 addresses long. This can be easily changed by the `defspace` macro.

Individual functional unit simulations can provide direct information to the user via the `pbbb` function. `Pbbb` stands for "Print Blow By Blow". If the symbol `blow-by-blow` is non-nil, `pbbb` prints its argument. If `blow-by-blow` is nil, `pbbb` is a no-op.

As an aid to running long simulations, the simulated sequencer (but not the real one) halts the machine when an attempt is made to execute location 0 of M0. In this way, a simulation can be started and allowed to run unattended.

8.5.3. Simulation Support Functions

The DIS simulator is controlled by means of several end-user functions. Besides `mclink`, several other steps must be taken to run a program in the simulator.

If it is desired to keep a machine-readable record of the run, `dribble` must be executed. Then, `final-status` can be set. The default value, `T`, means print a complete machine status readout at the end of each machine cycle. The value `short` means print only the address of the next instruction to be executed. A value of `nil` suppresses all end-of-cycle user messages.

The user can set a "stopwatch" of how many 100 nS cycles have been executed, by the global variable `cycle-counter`. `Cycle-counter` is incremented at the start of each cycle.

When all executables needed have been linked into the simulation memory, the user `cold-boots` the simulator. `Cold-boot` accepts a varying number of arguments. The first argument is the starting location to be executed (normally the load address of the main routine). All subsequent arguments to `cold-boot` are taken as arguments to the program being executed on the DIS simulator. A proper stack-frame is built, with the return pointer pointing to location 0. In this way, the simulation stops when the main function returns. In a real DIS machine, the main program would be a read-eval-print loop, and would never return.

`Cold-boot` is necessary to properly set up the instruction-fetch pipeline for proper routine execution. If instruction `n` were being executed, then instruction `n+1` is currently being output by `M0`, and instruction `n+2` is being addressed by the input port of `M0`. Hence, `cold-boot` must set up the proper output values of all functional units so that the DIS simulator "looks" like it has been running already.

Finally, the user can run the simulation by `disrun`. `Disrun` takes one argument—the maximum number of cycles to run. This is an upper limit. The simulation will stop when this limit expires, when the simulation branches to zero (main routine returns) or when the user hits control-C.

8.6. Testing the DIS Software

The DIS software was tested in three phases.

First, short assembly-language routines were written to exercise each part of the simulator. The resulting instruction traces were then hand-checked for correctness.

Once the simulator was reasonably debugged, we used it to check the output of the DIS compiler. The compiler itself emits commented assembly code, which can be hand-checked. The simulator was then used to execute the compiler output, and the instruction traces were hand checked.

Finally, the DIS simulator was used to execute actual routines, with correctness of results being the final objective. The DIS simulator running DIS compiler output was used to calculate some short expressions. The longest run was the calculation of $10!$, the result (3,628,800) took several hours, but was correct. Sections of the instruction traces were hand examined in these long runs, and the DIS simulator appears to do what it was designed to do, albeit very slowly.

Because the DIS simulator is written as a set of system-definition macros, defining memories, stacks, etc. it is highly unlikely that some essential part of the DIS design is not being exercised in some way by at least one of the test functions. For example, Memory 0 is never written to, but the same macro that defines Memory 0 is used to define Memory 1 and Memory 2 (which are often written to), so it is unlikely that there is a bug in the code that defines how M0 performs a write. If we consider that we have tested a code path that is invoked by macro by testing every path through that macro at least once in the system, there are no paths in the DIS software (simulator, compiler and optimizer) that have not been covered for testing purposes.

Chapter 9

Examples of DIS Machine Code

This section gives examples of DIS processor execution of several simple problems, as they would be executed in compiled code on the proposed design.

The code streams presented below are abstractions of the compiler result for easier human comprehension, with chunks of actual compiler output included where appropriate. Compiler generated output often has comments inserted- the compiler generates these comments as an aid to debugging the compiler.

The reader should remember that the instructions emitted by the compiler are incomplete in that they do not contain any module-idle or non-branch sequencer opcodes. Module-idle and sequencer opcodes are inserted after most other optimizations are complete.

Most of the included compiler/optimizer output is generated with the resource-merging phase turned off. The reason for this is that code manipulated by the resource merger is *very* obtuse. Where appropriate, resource-merged chunks of compiler output will be displayed with unmerged chunks, so the reader can see how the resource merger works.

Three notations are used for describing the state of a processor. The first notation is a stack notation; stack entries are separated by "||", with the top entry on the right. In this stack notation we will merge the contents of the pending call stack and the contents of the value stack, as though only one stack was used. The second notation is a microinstruction notation; everything between "[" and "]" is executed in one instruction phase. For example, 42: [mumble] indicates that the instruction at 42 is mumble . Lastly, actual compiler-generated code will be presented in the list notation:

```
(( (OP1 mumble) (OP2 bumble) (OP3 bruise) (OP7 33))  
  ((OP1 droll) (OP4 stumble) (OP5 gargle))  
  ((OP2 fiddle) (OP4 twiddle) (OP5 frobnicate)))
```

which is a list of three instructions, each instruction having several fields.

Several abbreviations are used to make the horizontal code less verbose and more understandable. The C language notation FOO++ is used as a shorthand for an operation that increments the register FOO. The arrow "←" represents data flow. Operations within the ALU are represented by their algebraic equivalent.

Some operations are understood to happen every cycle UNLESS one of the involved modules is explicitly directed to some other use. These assumed operations are the single incrementation of the next addressed location, the subsequent reloading of the current location from the next address bus, M0 instruction fetch, and command load from the M0 output. More formally:

SD1 ← B-SDO	address generator feedback
B-SDO ← SD1 + 1	next address is N + 1
MOVO ← B-SDO	MO fetches next instruction
SDO ← B-MDO	MO supplies fetched instructions

Because of the delayed branch and call, it should be noted that although an instruction indicates a transfer of control, that transfer does not take effect until after the next two instructions are executed. The compiler worries about this; the user should never see it.

Finally, we will often abbreviate the various instructions to a more easily readable form. Instead of specifying the prepare-to-call sequence (see above), we will use the words Pushcall, Executecall, and Popcall to indicate these in an understandable shorthand.

9.1. Addition of two literal values

We wish to add two literal values. These values are imbedded in the program. This is an unnatural program to execute more than once, but it provides a good example to start with:

(+ 2 3)

The stack would look like this

2	; push the 2
2 3	; push the 3
5	; pop and write the result.

A compiler might produce code such as:

```

20: [ VDO ← Immdata 2, Vpush 1]
21: [ VDO ← Immdata 3, Vpush 1]
22: [ ADO ← B-VDO, AD1 ← B-VD1, Add, Vpop 2]
23: [ VDO ← B-ADO, Vpush 1]

```

which returns a 5 on top of the stack and erases the previous arguments, and used 4 cycles. The actual compiler output for this (before optimization and default field insertion) is:

```

(((COMMENT "Pushing a CMI value onto VS") (VVO B-CMI)
 (VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
 (CMI 2))
(((COMMENT "Pushing a CMI value onto VS") (VVO B-CMI)
 (VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
 (CMI 3))
(((COMMENT "Adding top two elements of VS in ALU")
 (AVO B-VDO) (AV1 B-VD1) (ACO BOOLB) (AC1 SELAVO)
 (AC2 FMTFIX) (AC3 COMBADD) (AC4 SHIFTO))

```

```
((COMMENT "Now put result back on value stack")
(VVO B-ADO) (VCO SNORMAL) (VC1 SPOP1) (VC2 SWTOP)
(VC3 SPWRITE)))
```

Of course, this code fragment doesn't do any good to anyone- it just trails off without returning to anywhere. We could compile instead:

```
(defun z () (+ 2 3))
```

which could be compiled to:

```
20: [ VDO ← Immdata 2, Vpush 1]
21: [ VDO ← Immdata 3, Vpush 1]
22: [ ADO ← B-VDO, AD1 ← B-VD1, Add, Vpop 2]
23: [ VDO ← B-ADO, Vpush 1]
24: [ Saveresult ]
25: [ Returncall ]
```

The extra cycles at the end are needed to return to the caller. The compiler generates (before optimization):

```
((LOCATION Z)
 (COMMENT "Source code:" (DEFUN Z () (+ 2 3))))
((COMMENT "Pushing a CMI value onto VS") (VVO B-CMI)
 (VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
 (CMI 2))
((COMMENT "Pushing a CMI value onto VS") (VVO B-CMI)
 (VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
 (CMI 3))
((COMMENT "Adding top two elements of VS in ALU")
 (AVO B-VDO) (AV1 B-VD1) (ACO BOOLB) (AC1 SELAVO)
 (AC2 FMTFIX) (AC3 COMBADD) (AC4 SHIFTO))
((COMMENT "Now put result back on value stack")
 (VVO B-ADO) (VCO SNORMAL) (VC1 SPOP1) (VC2 SWTOP)
 (VC3 SPWRITE))
((COMMENT "Return to caller using RS data")
 (COMMENT "No branch permitted - return latency 2")
 (CRUSHMAX 2) (RESERVE SEQ VS RS) (MOV0 B-RDO)
 (MOC MREAD) (SVO B-MDO) (SV1 B-RDO) (SV2 B-RDO)
 (SCO 1) (SC1 1) (SC2 0) (SC3 0))
((COMMENT "No branch permitted - return latency 1")
 (CRUSHMAX 1) (SCO SEQNEXT) (VVO B-VDO) (VV1 B-RD1)
 (VCO SUBHEAD) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
 (RCO SNORMAL) (RC1 SPOP2) (RC2 SWIDLE)
 (RC3 SPWRITE))
((LOCATION ANONYMOUS)
 (COMMENT "labels prevent statement migration.")))
```

The 'instructions' which contain only comments and LOCATION fields will be

optimized into other instructions, and the two instruction latency at the tail can be optimized by overlapping. This translates to:

```

20: [ VDO ← Immdata 2, Vpush 1]
21: [ VDO ← Immdata 3, Vpush 1]
22: [ ADO ← B-VDO, AD1 ← B-VD1, Add, Vpop 2]
23: [ VDO ← B-ADO, Vpush 1, Returncall ]
24: [ Saveresult ]

```

The current compiler and optimizer (with resource-combination disabled) together produced the following code:

```

(((LOCATION Z)
  (COMMENT "Source code:" (DEFUN Z () (+ 2 3)))
  (COMMENT "Pushing a CMI value onto VS") (VVO B-CMI)
  (VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
  (CMI 2))
  ((COMMENT "Pushing a CMI value onto VS") (VVO B-CMI)
  (VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
  (CMI 3))
  ((COMMENT "Adding top two elements of VS in ALU")
  (AVO B-VDO) (AV1 B-VD1) (ACO BOOLB) (AC1 SELAVO)
  (AC2 FMTFIX) (AC3 COMBADD) (AC4 SHIFTO))
  ((COMMENT "Now put result back on value stack")
  (VVO B-ADO) (VCO SNORMAL) (VC1 SPOP1) (VC2 SWTOP)
  (VC3 SPWRITE)
  (COMMENT "Return to caller using RS data")
  (COMMENT "No branch permitted - return latency 2")
  (CRUSHMAX 2) (RESERVE SEQ VS RS) (MOVO B-RDO)
  (MOC MREAD) (SVO B-MDO) (SV1 B-RDO) (SV2 B-RDO)
  (SCO 1) (SC1 1) (SC2 0) (SC3 0))
  ((COMMENT "No branch permitted - return latency 1")
  (CRUSHMAX 1) (SCO SEQNEXT) (VVO B-VDO) (VV1 B-RD1)
  (VCO SUBHEAD) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
  (RCO SNORMAL) (RC1 SPOP2) (RC2 SWIDLE)
  (RC3 SPWRITE))
  ((LOCATION ANONYMOUS)
  (COMMENT "labels prevent statement migration.")))

```

where it can be seen the return instruction is being fired off while the result is still in the ALU. This particular case has saved us only one instruction (5 instead of 6), but in longer codes the optimizer has more possible overlapping to work with, especially when references to both the symbol table and the data memory occur near each other.

9.2. Addition of two variables

Of course, the case of adding two variables is pathological, as the compiler should (or might) at least notice it was a constant. Much more reasonable is addition of two variables. If we assume the variables are already on the stack:

```
(+ (.....) (.....) )
```

which codes as:

```
...<previous evaluation of arguments>
...
20: [ ADO ← B-VDO, AD1 ← B-VD1, Add ]
21: [ VDO ← B-ADO, Vpush 1, Returncall ]
22: [ Saveresult ]
```

This code fragment by itself is only moderately interesting. Let us compile a complete expression of the form:

```
(+ <local-arg> <global-var> )
```

More specifically, let us compile:

```
(defun f1 (x) (+ x y))
```

This would compile to something like:

```
20: [ Vsubhead-read ]
21: [ VDO ← B-VDO, Vpush 1 ]
22: [ M1 ← Immdata (hashloc of Y), M1Read ]
23: [ M2 ← B-MD1, M2Read ]
24: [ VDO ← B-MD1, Vpush 1 ]
25: [ ADO ← B-VDO, AD1 ← B-VD1, Add, Vpop 2 ]
26: [ VDO ← B-ADO, Vpush 1 ]
27: [ Returncall ]
28: [ Saveresult ]
```

The actual code generated by the compiler and optimizer (without resource merging) is:

```
((LOCATION F1)
 (COMMENT "Source code:" (DEFUN F1 (X) (+ X Y)))
 (COMMENT "Getting a local value stacked on VS onto B-VDO")
 (COMMENT "Address we want is RD1 + vv0 - sidle + 1")
 (VC1 8) (VV1 B-RD1) (VCO SUBHEAD) (VC2 SWIDLE)
 (VC3 SPIDLE)
 (COMMENT "Getting global value of symbol" Y)
 (CMI 1032)
 (COMMENT "Cycle M1 from CMI addr and get absolute addr")
```



```

(M1VO B-CMI) (M1C MREAD))
((COMMENT "Now we push B-VDO back for local copy on VS")
(VVO B-VDO) (VV1 B-VDO) (VCO SNORMAL) (VC1 SPUSH1)
(VC2 SWTOP) (VC3 SPWRITE))
((COMMENT "Take the pointer M1 gave us, put it on VS")
(VVO B-MD1) (VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP)
(VC3 SPWRITE))
((COMMENT "Adding top two elements of VS in ALU")
(AVO B-VDO) (AV1 B-VD1) (ACO BOOLB) (AC1 SELAVO)
(AC2 FMIFIX) (AC3 COMBADD) (AC4 SHIFTO))
((COMMENT "Now put result back on value stack")
(VVO B-ADO) (VCO SNORMAL) (VC1 SPOP1) (VC2 SWTOP)
(VC3 SPWRITE)
(COMMENT "Return to caller using RS data")
(COMMENT "No branch permitted - return latency 2")
(CRUSHMAX 2) (RESERVE SEQ VS RS) (MOVO B-RDO)
(MOC MREAD) (SVO B-MDO) (SV1 B-RDO) (SV2 B-RDO)
(SCO 1) (SC1 1) (SC2 0) (SC3 0))
((COMMENT "No branch permitted - return latency 1")
(CRUSHMAX 1) (SCO SEQNEXT) (VVO B-VDO) (VV1 B-RD1)
(VCO SUBHEAD) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
(RCO SNORMAL) (RC1 SPOP2) (RC2 SWIDLE)
(RC3 SPWRITE))
((LOCATION ANONYMOUS)
(COMMENT "labels prevent statement migration.")))

```

We see that not only were we able to overlap the return with the stacking of the final result, but we were able to overlap the fetches of the stack-based local variables with the memory-resident dynamic variables.

The compiler can be told to allow optimizations of resources such as noticing the two successive Vpushes (in the second and third instruction) could have been combined into a single Vpush2. The compiler then generates the output stream:

```

(((LOCATION F1)
(COMMENT "Source code:" (DEFUN F1 (X) (+ X Y)))
(COMMENT "Getting a local value stacked on VS onto B-VDO")
(COMMENT "Address we want is RD1 + vv0 - sidle + 1")
(VC1 8) (VV1 B-RD1) (VCO SUBHEAD) (VC2 SWIDLE)
(VC3 SPIDLE)
(COMMENT "Getting global value of symbol" Y)
(CMI 1024)
(COMMENT "Cycle M1 from CMI addr and get absolute addr")
(M1VO B-CMI) (M1C MREAD))
((COMMENT "Now we push B-VDO back for local copy on VS")
(VV1 B-VDO)
(COMMENT "Combining two vpush-1's into a vpush-2")
(VCO SNORMAL) (VC1 SPUSH2) (VC2 SWBOTH)
(VC3 SPWRITE)

```

```

(COMMENT "Take the pointer M1 gave us, put it on VS")
(VVO B-MD1))
((COMMENT "Adding top two elements of VS in ALU")
(AVO B-VDO) (AV1 B-VD1) (ACO BOOLB) (AC1 SELAVO)
(AC2 FMTFIX) (AC3 COMBADD) (AC4 SHIFTO))
((COMMENT "Now put result back on value stack")
(VVO B-ADO) (VCO SNORMAL) (VC1 SPOP1) (VC2 SWTOP)
(VC3 SPWRITE)
(COMMENT "Return to caller using RS data")
(COMMENT "No branch permitted - return latency 2")
(CRUSHMAX 2) (RESERVE SEQ VS RS) (MOV0 B-RDO)
(MOC MREAD) (SVO B-MDO) (SV1 B-RDO) (SV2 B-RDO)
(SC0 1) (SC1 1) (SC2 0) (SC3 0))
((COMMENT "No branch permitted - return latency 1")
(CRUSHMAX 1) (SC0 SEQNEXT) (VVO B-VDO) (VV1 B-RD1)
(VCO SUBHEAD) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
(RCO SNORMAL) (RC1 SPOP2) (RC2 SWIDLE)
(RC3 SPWRITE))
((LOCATION ANONYMOUS)
(COMMENT "labels prevent statement migration.")))

```

which buys us another 15 % in speed at the expense of human readability.

9.3. Nested Routine Calls

The compiler generates correct code for nested subroutine calls. Fortunately, LISP subroutine calls are always expressed as having the argument to a function being another function evaluation. We can use our call/return stacks to good advantage here (this is their intended purpose). The compiler simply places arguments on the value stack as needed and inserts executecalls when the argument list of a function has been completed.

For an example, suppose we wish to do a sum-of-products. Further, let us suppose that the arguments are mixed: some are literal, and some are symbols:

```
(+ (invoke 3 voodoo) (speak 23 skiddoo))
```

We will suppose that the functions "invoke" and "speak" are presupplied (maybe they do type-checking or work for integers, floats, and bignums interchangeably). From the stack point of view, this function looks like:

```

+
+ || invoke
+ || invoke || 3
+ || invoke || 3 || voodoo
+ || invoke || 3 || 99
+ || 297
+ || 297 || speak

```

```

+ || 297 || speak || 23
+ || 297 || speak || 23 || skiddoo
+ || 297 || 46
343

```

The code generated might look like:

```

50: [ Pushcall to "invoke" ]
51: [ VDO ← Immdata 3, Vpush 1 ]
52: [ M1DO ← Immdata voodoo, M1Read ]
53: [ M2DO ← B-MD1, M2Read ]
54: [ VDO ← B-MD2, Vpush 1 ]
55: [ Executecall ]
56: [ latency NOP ]
57: [ Pushcall to "speak" ]
58: [ VDO ← Immdata 23 ]
59: [ M1DO ← Immdata skidoo, M1Read ]
60: [ M2DO ← B-MD1, M2Read ]
61: [ VDO ← B-MD2, Vpush 1 ]
62: [ Executecall ]
63: [ latency NOP ]
64: [ ADO ← B-VDO, AD1 ← B-VD1, Add ]
65: [ VDO ← B-ADO, Vpush 1 ]
66: [ Returncall ]
67: [ Saveresult ]

```

The actual compiler/optimizer (no resource merger) output is:

```

((COMMENT "Preparing to call the function named " INVOKE)
(CMI 1040)
(COMMENT "Cycle M1 from CMI addr +1 offset and get absaddr")
(M1VO B-CMI) (M1V2 1) (M1C MREAD))
((COMMENT "Take the pointer M1 gave us, put it on PS with VSP")
(PVO B-MD1) (PV1 B-VSP) (PCO SNORMAL) (PC1 SPUSH2)
(PC2 SWBOTH) (PC3 SPWRITE)
(COMMENT "Pushing a CMI value onto VS") (VVO B-CMI)
(VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
(CMI 3))
((COMMENT "Getting global value of symbol" VOODOO)
(CMI 1048)
(COMMENT "Cycle M1 from CMI addr and get absolute addr")
(M1VO B-CMI) (M1C MREAD)
(COMMENT "Execute a prepared call")
(COMMENT "No branch permitted - call latency 2")
(CRUSHMAX 1) (RESERVE SEQ) (RVO B-SDO) (RV1 B-PD1)
(RCO SNORMAL) (RC1 SPUSH2) (RC2 SWBOTH)
(RC3 SPWRITE) (MOVO B-PDO) (MOC MREAD) (SVO B-MDO)
(SV1 B-PDO) (SV2 B-PDO) (SCO 1) (SC1 1) (SC2 0)
(SC3 0))

```

```

((COMMENT "Take the pointer M1 gave us, put it on VS")
(VVO B-MD1) (VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP)
(VC3 SPWRITE)
(COMMENT "No branch permitted - call latency 1")
(CRUSHMAX 1) (SCO SEQNEXT))
((LOCATION ANONYMOUS)
(COMMENT "labels prevent statement migration.")
(COMMENT "Preparing to call the function named " SPEAK")
(CMI 1056)
(COMMENT "Cycle M1 from CMI addr +1 offset and get absaddr")
(M1VO B-CMI) (M1V2 1) (M1C MREAD))
((COMMENT "Take the pointer M1 gave us, put it on PS with VSP")
(PVO B-MD1) (PV1 B-VSP) (PCO SNORMAL) (PC1 SPUSH2)
(PC2 SWBOTH) (PC3 SPWRITE)
(COMMENT "Pushing a CMI value onto VS") (VVO B-CMI)
(VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
(CMI 23))
((COMMENT "Getting global value of symbol" SKIDD00)
(CMI 1064)
(COMMENT "Cycle M1 from CMI addr and get absolute addr")
(M1VO B-CMI) (M1C MREAD)
(COMMENT "Execute a prepared call")
(COMMENT "No branch permitted - call latency 2")
(CRUSHMAX 1) (RESERVE SEQ) (RVO B-SDO) (RV1 B-PD1)
(RCO SNORMAL) (RC1 SPUSH2) (RC2 SWBOTH)
(RC3 SPWRITE) (MOVO B-PDO) (MOC MREAD) (SVO B-MDO)
(SV1 B-PDO) (SV2 B-PDO) (SCO 1) (SC1 1) (SC2 0)
(SC3 0))
((COMMENT "Take the pointer M1 gave us, put it on VS")
(VVO B-MD1) (VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP)
(VC3 SPWRITE)
(COMMENT "No branch permitted - call latency 1")
(CRUSHMAX 1) (SCO SEQNEXT))
((LOCATION ANONYMOUS)
(COMMENT "labels prevent statement migration.")
(COMMENT "Adding top two elements of VS in ALU")
(AVO B-VDO) (AV1 B-VD1) (ACO BOOLB) (AC1 SELAVO)
(AC2 FMTRFIX) (AC3 COMBADD) (AC4 SHIFTO))
((COMMENT "Now put result back on value stack")
(VVO B-ADO) (VCO SNORMAL) (VC1 SPOP1) (VC2 SWTOP)
(VC3 SPWRITE)))

```

Ten cycles are used, not counting time spent in the functions INVOKE and SPEAK. Each function will take at least 3 cycles (300 nanoseconds) so the minimum timing for this program is 1.6 microseconds- quite a reasonable showing. Compared to a VAX, this is blazing speed; a VAX takes about twice as long just to execute a subroutine call *instruction*, not even counting the argument transfer or return instruction.

9.4. Recursive Functions

In many applications, a simple recursive function is much easier to program than the iterative function. This design supports iteration very nicely, thanks to the pending/return stack mechanism.

At the risk of being conventional, we shall use the factorial function as an example. Factorial is defined in LISP as:

```
(defun factorial (n)
  (cond
    ((equal n 1) 1)
    (t (times n (factorial (sub1 n))))))
```

In English, this function definition says that factorial of x is 1 if x is one, else it is x times factorial of x minus 1. To see what the stack looks like during a factorial, let us observe factorial of two, assuming the compiler "open-codes" the cond function. Special attention must be paid to the conditional function in any case, because it does NOT get evaluated arguments- cond may or may not evaluate it's own arguments.

```
factorial ||2
factorial ||2 ||equal
factorial ||2 ||equal ||2
factorial ||2 ||equal ||2 ||1
factorial ||2 ||nil
factorial ||2 ||t
factorial ||2 ||times
factorial ||2 ||times ||2
factorial ||2 ||times ||2 ||factorial
factorial ||2 ||times ||2 ||factorial ||sub1
factorial ||2 ||times ||2 ||factorial ||sub1 ||2
factorial ||2 ||times ||2 ||factorial ||1
factorial ||2 ||times ||2 ||factorial ||1 ||equal
factorial ||2 ||times ||2 ||factorial ||1 ||equal ||1
factorial ||2 ||times ||2 ||factorial ||1 ||equal ||1 ||1
factorial ||2 ||times ||2 ||factorial ||1 ||T
factorial ||2 ||times ||2 ||1
factorial ||2 ||2
2
```

Because the factorial function is complex, the number of elements on the stack when the factorial function has completed may not be correct; exactly as many items should be on the stack as when "factorial" was invoked. We can insure this by using the Returncall instruction, which reloads the value stack pointer from the stored subhead pointer. The reloading occurs during the current instruction phase, and we therefore have two instructions to use this local "abberation" to our advantage in properly returning a value, no matter how badly the stack may have

been mismanaged. With a debugged compiler this situation should not occur, but "firewalls" in the code are always good practice (especially in handmade code).

Assuming that the functions "equal", "times" and "sub1" are available for calling, the compiler might generate the following code:

```

30: [ Build-fake-return-frame]
31: [ Pushcall to "equal" ]
32: [ Vsubhead-read 1 ]
33: [ VDO ← B-VDO ]
34: [ VDO ← Immdata 1, Vpush 1 ]
35: [ Executecall ]
36: [ latency NOP ]
37: [ Branch-on-NIL to "Further" ]
38: [ latency NOP ]
39: [ latency NOP ]
40: [ VDO ← Immdata 1, Voverwrite-top ]
41: [ Fakereturn ]
42: [ latency NOP ]
43: FURTHER: [ Pushcall to "times" ]
44: [ Vsubhead-read 1 ]
45: [ VDO ← B-VDO, Vpush 1 ]
46: [ Pushcall to "factorial" ]
47: [ Pushcall to "sub1" ]
48: [ Vsubhead-read 1 ]
49: [ VDO ← B-VDO ]
50: [ Executecall ]
51: [ latency NOP ]
52: [ Executecall ]
53: [ latency NOP ]
54: FAKERETURN: [ Saveresult ]
55: [ Returncall ]
56: [ latency NOP ]

```

A number of optimizations can be done on this program. The current compiler can inline-code such primitives as sub1 (Common LISP 1-), as well as times (Common LISP *). If we also take EQ to be adequate (two equal fixnums are guaranteed to be EQ), we get the following program from the compiler/optimizer (no resource merger):

```

(((LOCATION FACT)
  (COMMENT "Source code:"
    (DEFUN FACT (N)
      (COND ((EQ N 1) 1)
            (T (* N (FACT (- N 1)))))))
  (COMMENT "Creating a fake return frame")
  (CMI COND-EXIT-3753) (RVO B-CMI) (RV1 B-RD1)
  (RCO SNORMAL) (RC1 SPUSH2) (RC2 SWBOTH)
  (RC3 SPWRITE))

```

```

((COMMENT "Getting a local value stacked on VS onto B-VDO")
  (COMMENT "Address we want is RD1 + vv0 - sidle + 1")
  (VC1 8) (VV1 B-RD1) (VCO SUBHEAD) (VC2 SWIDLE)
  (VC3 SPIDLE))
((COMMENT "Now we push B-VDO back for local copy on VS")
  (VVO B-VDO) (VV1 B-VDO) (VCO SNORMAL) (VC1 SPUSH1)
  (VC2 SWTOP) (VC3 SPWRITE))
((COMMENT "Pushing a CMI value onto VS") (VVO B-CMI)
  (VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
  (CMI 1))
((COMMENT "Checking VS and VS-1 for EQness")
  (COMMENT "No branch permitted - skip latency 3")
  (RESERVE SEQ ALU VS) (CRUSHMAX 3) (AVO B-VDO)
  (AV1 B-VD1) (ACO BOOLXOR) (SV1 B-SD0) (SV2 B-SD0)
  (SCO SEQNEXT) (SC1 SEQSKIP) (SC2 SEQMASKZERO))
((COMMENT "We pop the VS, and overwrite a T onto it. ")
  (COMMENT "No branch permitted - skip latency 2")
  (RESERVE SEQ ALU VS) (CRUSHMAX 2) (VVO B-CMI)
  (VCO SNORMAL) (VC1 SPOP1) (VC2 SWTOP) (VC3 SPWRITE)
  (CMI 1000) (CV1 1) (CV2 1))
((COMMENT "Here we sit and wait. Nothing to do.")
  (COMMENT "No branch permitted - skip latency 1")
  (RESERVE SEQ ALU VS) (CRUSHMAX 1))
((COMMENT "This instruction may or may not be executed.")
  (COMMENT "No branch permitted, skip latency 0 ??? ")
  (COMMENT "It's here we maybe push a NIL")
  (RESERVE SEQ ALU VS) (CRUSHMAX 1) (CMI 0) (CV1 1)
  (CV2 0) (VVO B-CMI) (VCO SNORMAL) (VC1 SIDLE)
  (VC2 SWTOP) (VC3 SPIDLE))
((LOCATION ANONYMOUS)
  (COMMENT "labels prevent statement migration.")
  (COMMENT "Branch-if-null to " COND-NEXT-3754)
  (CMI COND-NEXT-3754)
  (COMMENT "No branch permitted - cond branch latency 3")
  (CRUSHMAX 3) (RESERVE SEQ) (AVO B-VDO) (ACO BOOLA)
  (SV1 B-SD0) (SV2 B-CMI) (SCO SEQNEXT) (SC1 SEQSAME)
  (SC2 SEQMASKNIL))
((COMMENT "No branch permitted - cond branch latency 2")
  (CRUSHMAX 2)
  (COMMENT "if branching, M0 sees new address this cycle")
  (SCO SEQNEXT))
((COMMENT "No branch permitted - cond branch latency 1")
  (COMMENT "if branching, M0 returned new instr. stream to seq")
  (CRUSHMAX 1) (SCO SEQNEXT))
((LOCATION ANONYMOUS)
  (COMMENT "labels prevent statement migration.")
  (COMMENT "Popping VS once") (VCO SNORMAL)
  (VC1 SPOP1) (VC2 SWIDLE) (VC3 SPWRITE)

```

```

(COMMENT "Return to faked caller using RS data")
(COMMENT "No branch permitted - return latency 2")
(CRUSHMAX 1) (RESERVE SEQ RS) (MOV0 B-RD0)
(MOC MREAD) (SVO B-MD0) (SV1 B-RD0) (SV2 B-RD0)
(SC0 1) (SC1 1) (SC2 0) (SC3 0))
((COMMENT "Pushing a CMI value onto VS") (VVO B-CMI)
(VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
(CMI 1)
(COMMENT "No branch permitted - return latency 1")
(CRUSHMAX 1) (SCO SEQNEXT) (RCO SNORMAL) (RC1 SPOP2)
(RC2 SWIDLE) (RC3 SPWRITE))
((LOCATION ANONYMOUS)
(COMMENT "labels prevent statement migration.")
((LOCATION COND-NEXT-3754)
(COMMENT "Source code:"
"Start of the next clause here")
(COMMENT "Popping VS once") (VCO SNORMAL)
(VC1 SPOP1) (VC2 SWIDLE) (VC3 SPWRITE)
(COMMENT "Preparing to call the function named "
FACT)
(CMI 1016)
(COMMENT "Cycle M1 from CMI addr+1 and get absolute addr")
(M1VO B-CMI) (M1V2 1) (M1C MREAD))
((COMMENT "Getting a local value stacked on VS onto B-VDO")
(COMMENT "Address we want is RD1 + vv0 - sidle + 1")
(VC1 8) (VV1 B-RD1) (VCO SUBHEAD) (VC2 SWIDLE)
(VC3 SPIDLE))
((COMMENT "Now we push B-VDO back for local copy on VS")
(VVO B-VDO) (VV1 B-VDO) (VCO SNORMAL) (VC1 SPUSH1)
(VC2 SWTOP) (VC3 SPWRITE))
((COMMENT "Take the pointer M1 gave us, put it on PS with VSP")
(PVO B-MD1) (PV1 B-VSP) (PCO SNORMAL) (PC1 SPUSH2)
(PC2 SWBOTH) (PC3 SPWRITE)
(COMMENT "Getting a local value stacked on VS onto B-VDO")
(COMMENT "Address we want is RD1 + vv0 - sidle + 1")
(VC1 8) (VV1 B-RD1) (VCO SUBHEAD) (VC2 SWIDLE)
(VC3 SPIDLE))
((COMMENT "Now we push B-VDO back for local copy on VS")
(VVO B-VDO) (VV1 B-VDO) (VCO SNORMAL) (VC1 SPUSH1)
(VC2 SWTOP) (VC3 SPWRITE))
((COMMENT "Pushing a CMI value onto VS") (VVO B-CMI)
(VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
(CMI 1))
((COMMENT "Subtract top two elements of VS in ALU")
(AVO B-VDO) (AV1 B-VD1) (ACO BOOLB) (AC1 SELAVO)
(AC2 FMTFIX) (AC3 COMBSUBTRACT) (AC4 SHIFTO)
(COMMENT "Execute a prepared call")
(COMMENT "No branch permitted - call latency 2")

```



```

(CRUSHMAX 1) (RESERVE SEQ) (RVO B-SDO) (RV1 B-PD1)
(RCO SNORMAL) (RC1 SPUSH2) (RC2 SWBOTH)
(RC3 SPWRITE) (MOVO B-PDO) (MOC MREAD) (SVO B-MDO)
(SV1 B-PDO) (SV2 B-PDO) (SCO 1) (SC1 1) (SC2 0)
(SC3 0))
((COMMENT "Now put result back on value stack")
(VVO B-ADO) (VCO SNORMAL) (VC1 SPOP1) (VC2 SWTOP)
(VC3 SPWRITE)
(COMMENT "No branch permitted - call latency 1")
(CRUSHMAX 1) (SCO SEQNEXT))
((LOCATION ANONYMOUS)
(COMMENT "labels prevent statement migration.")
(COMMENT "Multiplying top two elements of VS in ALU")
(AVO B-VDO) (AV1 B-VD1) (ACO BOOLB) (AC1 SELAVO)
(AC2 FMIFIX) (AC3 COMBMULTIPLY) (AC4 SHIFTO)
(COMMENT "Return to faked caller using RS data")
(COMMENT "No branch permitted - return latency 2")
(CRUSHMAX 1) (RESERVE SEQ RS) (MOVO B-RDO)
(MOC MREAD) (SVO B-MDO) (SV1 B-RDO) (SV2 B-RDO)
(SCO 1) (SC1 1) (SC2 0) (SC3 0))
((COMMENT "Now put result back on value stack")
(VVO B-ADO) (VCO SNORMAL) (VC1 SPOP1) (VC2 SWTOP)
(VC3 SPWRITE)
(COMMENT "No branch permitted - return latency 1")
(CRUSHMAX 1) (SCO SEQNEXT) (RCO SNORMAL) (RC1 SPOP2)
(RC2 SWIDLE) (RC3 SPWRITE))
((LOCATION ANONYMOUS)
(COMMENT "labels prevent statement migration.")
(COMMENT "Pushing a CMI value onto VS") (VVO B-CMI)
(VCO SNORMAL) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
(CMI 1024)
(COMMENT "Popping a fake return frame off of RS")
(RCO SNORMAL) (RC1 SPOP2) (RC2 SWIDLE)
(RC3 SPWRITE))
((LOCATION COND-EXIT-3753)
(COMMENT "Source code:" "Where the cond exits")
(COMMENT "Return to caller using RS data")
(COMMENT "No branch permitted - return latency 2")
(CRUSHMAX 2) (RESERVE SEQ VS RS) (MOVO B-RDO)
(MOC MREAD) (SVO B-MDO) (SV1 B-RDO) (SV2 B-RDO)
(SCO 1) (SC1 1) (SC2 0) (SC3 0))
((COMMENT "No branch permitted - return latency 1")
(CRUSHMAX 1) (SCO SEQNEXT) (VVO B-VDO) (VV1 B-RD1)
(VCO SUBHEAD) (VC1 SPUSH1) (VC2 SWTOP) (VC3 SPWRITE)
(RCO SNORMAL) (RC1 SPOP2) (RC2 SWIDLE)
(RC3 SPWRITE))
((LOCATION ANONYMOUS)
(COMMENT "labels prevent statement migration.")))

```

If we enable resource merging in the compiler, the resulting program becomes two instructions faster- about 10 % faster.

9.5 Parallel code execution

For an example of parallel code execution, we shall program a simple AI program; a chess playing program using a parallel searching algorithm. To make the program parallelism more easily seen by the reader, we will not use any speedups such as alpha-beta pruning. The “uninteresting” parts of the program will not be described here; we will assume that the static game evaluator and the plausible move generator are already written.

The algorithm used is described recursively. The static game evaluator function “points” gives an evaluation of the current board statically, from the point of view of the white player, without doing any look-ahead. The plausible move generator “goodmoves” generates a variable-length list of good moves for the white player, based upon some board configuration. The “changesides” function creates a board description in which white and black have exchanged pieces (although the configuration generated often cannot appear in a real game, the exchanged board is used by the static evaluator and the plausible move generator). “Board-after-move” returns a board and a move where the move has been made to the board. “Bestmove” takes a list of moves and their associated point values, and returns the move with the highest point value.

Of course, the static evaluator and move generator, and move operator could have been written to accept a second argument “whose-move” but that would unnecessarily complicate the program. We need a recursive counter to tell the program when to stop exploring the tree any deeper. This counter counts game ply (a ply is one move by either player, sometimes called a half-turn).

For a more detailed description of inter (as opposed to *intraprocessor* parallelism), the reader is referred to R. Shane’s thesis (in progress).

The LISP code for the interesting part of this program would look something like:

```
(defun goodmove (board) .....)  
(defun points (board) .....)  
(defun changesides (board) .....)  
(defun board-after-move (board move) .....)  
      ; we can use changesides to get the  
      ; opponent’s best move, as:  
(defun black-best-move (board ply)  
  (white-best-move (changesides (board ply))))  
      ; now the recursive function for exploring  
      ; the game move tree  
(defun white-best-move (board ply)  
  (cond
```

```

(equal ply 1)
  (best (mapcar
    (getd 'points)
    (goodmoves board))))
(t (best (mapcar
  (getd 'points)
  (goodmoves
    (board-after-move
      board
      (black-best-move
        board
        (1- ply)))))))
(defun best (moves-and-points)          ; get move part of
  (car (bestmove (moves-and-points)    ; best pair
(defun bettermove (mp1 mp2)
  (cond
    ((greaterp
      (car mp1)
      (car mp2)) mp1)
    (t mp2)))
(defun bestmove (moves-and-points)
  (cond
    ((null (cdr moves-and-points))
      (car moves-and-points))          ; return move
    (t
      (bettermove
        (car moves-and-points)
        (bestmove (cdr moves-and-points))))))

```

The interesting part of the program is the mapcar in "whites-best-move". On a conventional processor, the implied iterative path is used, but since we are dealing with a multiprocessor, we may take advantage of any unused processor and speed our execution.

```

...< previous program code >
...
...< the stack state currently: >
...< T is a pointer to the reserved result cell>
...< T-1 is a pointer to the data ("goodmoves") >
...< PS has a pointer to the "points" function >
...< and the subhead pointer points to T-2 >
...
100: [ SeqPRI ← ::200, SeqSEC ← :SC0, SeqMASK ← T-OFVL ]
101: [ ... ]
102: [ ... ]
103: [ ExecuteCall ] ; Task-Overflow true- too many tasks
104:
.....< other code goes here to run function locally>

```

```

...
...
...< now the parallel mapcar code >
...
200: [ PushLocalVar GOODMOVES]
201: [ CDR VTOP ]
202: [ PushCall CONS ]
203: [ PushImmed NIL ]
204: [ ExecuteCall ]
205: [...] ;Now we have a pointer to a free cons cell...
206: [ NetTaskOut, PopPending2 ]
207: [ PushCall POINTS ]
208: [ PushLocalVar GOODMOVES ]
209: [ CAR VTOP ]
210: [ ExecuteCall ]
...
...<and we go on to execute "points" on the car of >
...<the data, just as if we were using a single proc. >
...

```

We should note that the compiler was aware that "points" was an applicative function, and could be executed in parallel. Of course, some processor must "watch" for completion of the mapcar; this task would normally be allotted to the outside function. Code to continue this watch is not shown here, but a simple walking down the list of results, waiting for each result word to be written (and thereby altered from the flag "empty" value supplied by the allocate-cons function) is sufficient. More complex schemes with other advantages are possible.

Chapter 10

Tests of the DIS architecture

To determine just how good the DIS architecture is for LISP execution, a series of benchmark programs was run on the DIS simulator, and on several real computers. The benchmark suite is a combination of problems from the Gabriel report [Gabriel 1984] and several additional problems to explore parts of LISP that I felt were not well exercised by Gabriel.

The benchmark suite was actually run on the DIS simulator in cases where the program was expected to complete in a reasonable time. Because the simulator is so slow (about 200 DIS instructions/hour when full statistics are generated), some programs could not be run. Instead, path lengths through the compiled and optimized programs were evaluated by hand. Path repetition counts (from the Gabriel report) were used to calculate the time to evaluate the function. If it was anticipated that it would take more than a week of simulation time, the program was not run on the simulator.

The other machines which ran the benchmark suite were a Symbolics 3670 with the optional instruction prefetch unit, a Texas Instruments Explorer (courtesy Texas Instruments), an AI VAXstation (courtesy Digital Equipment Corp), and a Cray-1 (courtesy RPI Plasma Dynamics Laboratory and Lawrence Livermore National Laboratory).

First we will describe the benchmarked machines, then the conditions of the benchmark. Finally, we will describe the results of the benchmarks.

10.1 Benchmarked Machine Implementations

The Cray-1 uses the fastest technology of these four systems, being built exclusively of ECL. The Cray-1 has a base 12.5 nanosecond cycle time. There are no MSI or LSI parts in a Cray-1, all functions being implemented in SSI gates. The Cray-1 is a three-quarter arc of a cylinder about 6 1/2 feet tall and 5 feet in diameter (not counting power supplies in the base). The particular Cray-1 tested was equipped with 4 megawords (32 megabytes) of memory.

The Symbolics 3670 is implemented in a mix of TTL and ECL, with some of the circuitry in custom MSI gate arrays. The 3670 processor (without memory) has seven boards in it, each board being about 14" by 24".

The TI Explorer is similar to the MIT CADR machines (also the predecessor of the Symbolics 3600 series). The Explorer uses a large number of custom gate-array chips to make it possible to place the entire processor on one 14 x 24" board.

The AI VAXstation is an implementation of most of the DEC VAX architecture on a single chip (in CMOS). A second chip is the floating point unit (unused by the

benchmarks). Rarely used VAX instructions are implemented by software emulation. The VAXstation CPU card is about 6" by 9".

10.1.1. DIS Machine Simulator Assumptions

Because the DIS machine tested was only a simulation and not actual hardware, it is reasonable to discuss what assumptions are embodied in the simulator.

The simulator assumed a single DIS processor. The DIS processor was a very small machine, with only 1000 words of memory in each memory unit (M0, M1, and M2), and only 2000 words of memory in each stack unit (value, pending, return, and binding). All memory and stack units had cache disabled.

Like the Symbolics and the TI Explorer, the DIS machine did not have garbage collection enabled. Like the Symbolics and Explorer, if consing memory became exhausted, the DIS system would have crashed. This is perhaps unreasonable, but most Symbolics and TI users run their machines this way anyway.

The DIS simulator times assumed a 20 nS crossbar and an 80 nS functional unit cycle time, for an overall cycle time of 100 nanoseconds (and allowing functional units to have an extra 20 nanoseconds of internal use time overlapping crossbar transfer time. Because the compiler doesn't generate any instructions that need cycle-stretching (like fetch-and-add) on the benchmark set, taking 100 nanoseconds as the machine cycle time is a worst-case estimate. The DIS machine might be able to cycle faster on some of the instructions in the benchmark, but we will ignore this speedup.

10.2. Testing Environment

The Symbolics 3670 ran Release 7 of ZetaLISP, the TI Explorer ran Version 1 of the Explorer operating system, the AI VAXstation ran Version 4.4 of VMS and Version V2.0 VAX Common LISP, and the Cray ran a LLNL-modified Cray-optimized version of PSL, version 3.2. The DIS compiler generated code for the DIS simulation.

The Cray PSL did not support the `let` special form correctly (it appears in the manual but the PSL system did not recognize it). Other than that, very little trouble was encountered in moving the benchmarks from one machine to another.

10.2.1. Garbage Collection Time Not Included

On the TI Explorer, Symbolics, and DIS machine, garbage collection was turned off. The VAXstation was commanded to notify the console whenever a garbage collection was invoked. It was possible to run the VAX benchmarks without encountering a garbage collection if the LISP environment was mostly empty when the benchmark was started. The Cray PSL system did need to do garbage collections, but since the PSL system automatically tallies the time spent in garbage collection, this time was subtracted out from the times shown below.

10.2.2. Paging and I/O Time Minimized but Included

Both the Symbolics and the TI Explorer had sufficient physical memory to contain the entire benchmark. The VAXstation LISP was configured with a 1.5 megabyte user area, and hence could also run an entire benchmark without paging. The Cray was being shared by several hundred other users, so it cannot be determined to what extent the Cray swapped in and out due to other user's needs for memory (nor is it likely that a Cray will be available to run in single-user mode).

The TI Explorer and Symbolics 3670 are single-user machines and hence did not have any significant competing processes. As no disk I/O was observed (via the run-bars) during the benchmarks, it is unlikely that any network server process interrupts nor virtual memory page faults were encountered during the benchmark runs.

The AI VAXstation was also run in single-user mode. The per-process accounting and realtime monitors showed that the Common LISP system did not page during the benchmarks.

The DIS machine, even in the restricted configuration described above, had sufficient memory to run the benchmarks without paging.

10.2.3. Runtime Data Type-Checking Disabled

Type-checking of data was disabled on all machines, if possible. The TI Explorer and Symbolics systems support type-checking in the hardware and microcode. The VAX LISP, Cray PSL, and DIS compilers were both set to generate non-type-checking (safety 0) code.

10.2.4. Enabling of Compiler Optimizations

The TI Explorer, Symbolics, and DIS systems always operate at full optimization, so no changes were made. VAX LISP was set for maximum optimization by `speed:3`, `safety:0`, PSL by `*R2I`, `*NOLINKE` and `*ORD` switch settings. This enabled conversion of tail-recursion to iterative branching on both the VAX and the Cray.

In summary:

Page-faulting was suppressed to the greatest extent possible.

Remaining page-fault time is included

Run-time data type-checking was turned off (if possible)

Maximum compiler optimization was turned on

Garbage collection time is subtracted out

10.3. The Benchmark Suite

The benchmark suite is composed of six different recursive LISP functions. Four are taken from the Gabriel report, and two are functions I added to explore deeper recursion and consings other than in the normal style.

The four Gabriel benchmarks are TAK, STAK, TAKL, and RDIV. TAK, STAK, and TAKL are based on the Takeuchi function, modified by John McCarthy.

Here are the definitions of the Gabriel-Takeuchi benchmarks. This is the actual text used, including line-feeds at the appropriate places. The *only* modifications done were for the Cray PSL system; subtract one is not supported directly. Instead, the form `(plus x -1)` is used.

```
; TAK - from Gabriel report
; TAK - invoke with (tak 18 12 6)
(defun tak (x y z)
  (cond
    ((not (< y x)) z)
    (t
     (tak
      (tak (1- x) y z)
      (tak (1- y) z x)
      (tak (1- z) x y))))))
;
;
; TAKL - from Gabriel report
; For TAKL, x=list of 18, y=list of 12, z=list of 6
(defun takl (x y z)
  (cond
    ((not (shorterp y x))
```



```

        z)
      (t
        (takl
          (takl (cdr x) y z)
          (takl (cdr y) z x)
          (takl (cdr z) x y))))
;
;
;   STAK - from Gabriel Report
; (stak 18 12 6)
(defun stak ()
  (cond
    ((not (< y x))z)
    (t
      (let ((x (let ((x (1- x))
                    (y y)
                    (z z))
                (stak)))
            (y (let ((x (1- y))
                    (y z)
                    (z x))
                (stak)))
            (z (let ((x (1- z))
                    (y x)
                    (z y))
                (stak))))
        (stak))))
;
;

```

All three of these Takeuchi functions recurse 63,609 times when invoked with the given arguments. Of that, 47,706 invocations of 1- or cdr will be done, meaning that in 15,902 cases, the "long" path is taken. The "short" path is taken 47,707 times. From this and the length of the compiled code paths it is possible to calculate how long the DIS architecture would take on the TAK family of functions.

The other Gabriel benchmark is the recursive division of a list into two lists. This is essentially a test of consing in the "forward" order (again, this is the exact text used):

```

;
;   RDIV2 (page 135 gabriel)
; invoke on a list of 200 elements.
(defun rdiv2 (x)
  (cond
    ((null x) nil)
    (t (cons (car 1) (rdiv2 (cdr (cdr x)))))))
;

```

This benchmark tests the ability of a system to cons in the normal way, with little stack manipulation required.

Because the Gabriel benchmarks lacked tests of deep recursion (with the attendant chances for cache flushing), and also lacked routines which would require a large amount of stack manipulation, I added two benchmarks of my own. These benchmarks are the factorial function, and a modification of `rdiv2` to make it cons in an "un-normal" way. This new function, `nrev`, forms a list by producing inverted cons cells. This un-normal consing requires the LISP system to manipulate the stack in nonstandard (but legitimate) ways.

The two new benchmarks are defined as follows (Cray PSL used (plus `n - 1`) instead of `(- n 1)`, otherwise this is the exact text used):

```

;
;                               Factorial
(defun fact (n)
  (cond
    ((eq n 1) 1)
    (t
     (* n (fact (- n 1))))))
;
;                               NREV
;                               invoke on a list of 200 elements.
(defun nrev (x)
  (cond
    ((null (cdr x))
     (x))
    (t
     (cons
      (nrev (cdr x))
      (car x)))))
;

```

10.4. Benchmark Results

The statistics reported below are average times needed to execute one pass through the function, when running the problem as stated above. For example, we know `tak` is executed 63,609 times, with 47,707 of the repetitions through the "fast" path. The numbers reported below are the weighted averages of the path lengths for DIS evaluation, and the measured CPU time divided by number of invocations for the other machines.

My original intention was to use the measured values published by Gabriel as a starting point. However, I was unable to duplicate many of his results. My errors varied from a factor of ten (Cray-1 running `tak`) to very good correspondence (3600 + IFU machine family running `tak`, `tak1`, and `stak`, Cray-1 running `rdiv` (.6 sec vs .74 sec)). In order to provide consistent timings, I have used my measured

values exclusively. If they have errors, at least the errors will be consistent with the calculated values for the DIS machine and the ratios of the performance values should remain fairly constant.

10.4.1. Method of Obtaining Timings

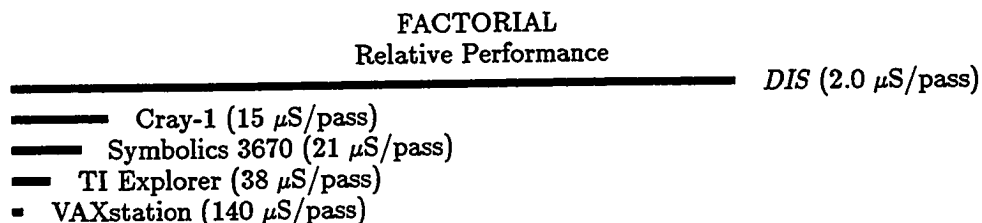
Each machine was "trusted" to the extent of believing the internal system clock. This has the advantage that the reflex speed of the observer is not a factor in obtaining timings. The disadvantage is that on some machines, the system clock "charges" the user for time spent in the operating system on the user's behalf (paging, swapping, etc.). The Cray-1 possibly charged swap time to the LISP process, but if so, the swapping costs were very consistent.

In general, I ran each benchmark three times, and observed the results. If the timings came within 5 % of each other (after subtracting GC time), I accepted the results as valid. The repeatability of the timings was on the order of 2 % or better. In only one case did this not occur- it was due to a typographical error in my input. When I reran that set of benchmarks, the timings came out within the 2 % boundary. The timings of the TI Explorer and Symbolics were extremely repeatable, the jitter being a few milliseconds (or 1/60 second counts) in a 21-second run.

10.4.2. Measured Results

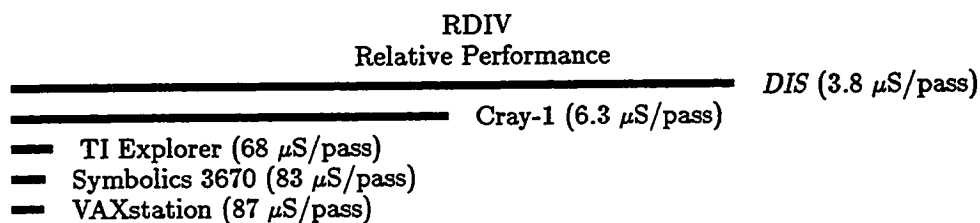
The DIS architecture showed strongly superior performance compared to all of the other machines tested, including the Cray-1. We will assume the cycle time of the DIS memories is 100 nanoseconds. Overall speed of the DIS design were on the order of twice to three times the Cray-1, running compiled LISP. The other machines (3670, Explorer, VAX) were usually orders of magnitude slower than the DIS and the Cray.

Each bar graph below indicates relative performance; so a longer bar indicates a faster system. The number of microseconds at the end of each bar indicates how long (in microseconds) it took that particular system to execute one pass of that particular benchmark.

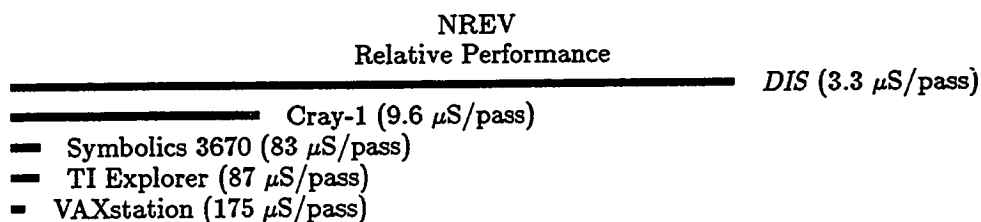


For the factorial function, as long as the LISP bignum math packages were not invoked, the speed leader was the DIS machine, at 2.0 μS. The Cray-1 turned in various times (depending on depth of recursion causing cache flushes) of 15 μS all

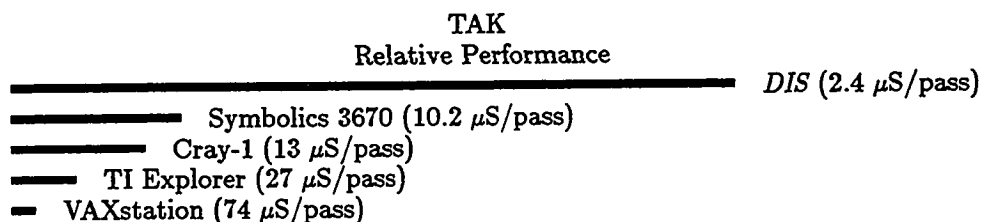
the way up to 41 μ S. The Symbolics 3670 was better than half the speed of the best Cray speed at 21 μ S. The TI Explorer did 38 μ S, and the AI VAXstation brought up the rear at 140 μ S. This benchmark shows the best timings of DIS versus Cray, with DIS being over twenty times faster for deep recursion and lots of arithmetic pipeline flushes. Even in the worst case for fact, the DIS architecture did better than seven times better than the Cray.



Rdiv (normal consing) showed timings of 3.8 μ S for the DIS, 6.3 μ S for the Cray, the Explorer at 68 μ S, the 3670 at 83 μ S, and the VAXstation at 87 μ S. This benchmark is the best showing of the Cray against DIS, with the DIS machine only 1.65 times faster.



Nrev (which does "un-normal" consing) turned in times of 3.3 μ S. for the DIS, 9.6 μ S for the Cray, 83 μ S for the 3670, 87 μ S for the Explorer, and 175 μ S for the VAXstation. It seems that the un-normal consing perturbed Cray PSL and VAX Common LISP far more than the stack-based DIS, 3600, and explorer.



Tak, which does fixnum math, showed 2.4 μ S for the DIS, 10.2 μ S for the 3670, 13 μ S for the Cray, 27 μ S for the Explorer, and 74 μ S for the VAXstation. I expect this poor showing for the Cray was due to repeated flushing of the arithmetic pipeline.

TAKL
Relative Performance

<hr style="width: 100%;"/>	<i>DIS</i> (8.2 μ S/pass)
<hr style="width: 100%;"/>	Cray-1 (23 μ S/pass)
<hr style="width: 100%;"/>	TI Explorer (93 μ S/pass)
<hr style="width: 100%;"/>	Symbolics 3670 (117 μ S/pass)
<hr style="width: 100%;"/>	VAXstation (630 μ S/pass)

Takl, which uses cdr instead of arithmetic to subtract, did not parallel tak for timings. The actual values found were 8.2 μ S for the DIS machine, 23 μ S for the Cray, 93 μ S for the Explorer, 117 μ S for the 3670, and 630 μ S for the VAXstation. This poor showing for the 3670 is most likely due to a combination of cache flushes (due to the repeated calls to shorterp and length), and the loss of "short-constant" usage in going from fixnums to lists.

STAK
Relative Performance

<hr style="width: 100%;"/>	<i>DIS</i> (3.4 μ S/pass)
<hr style="width: 100%;"/>	Symbolics 3670 (43 μ S/pass)
<hr style="width: 100%;"/>	TI Explorer (82 μ S/pass)
<hr style="width: 100%;"/>	VAXstation (157 μ S/pass)

Stak, which uses dynamically scoped global variables instead of Common LISP style lexical and lambda-bound variables, Timings found were 3.4 μ S for the DIS architecture, 43 μ S for the 3670, 82 μ S for the Explorer, and 157 μ S for the VAXstation. The Cray could not run this benchmark due to a system software problem with dynamically scoped variables in Cray PSL.

Chapter 11

Conclusions and Further Research

This thesis shows that it is possible to execute LISP in a multiple functional unit, horizontal microcode environment, at very high speeds. To achieve these high speeds, the applicative nature of LISP is used to automatically find parallelism and map that parallelism onto a machine model with multiple functional units.

The high speeds are achievable for a wide class of functions, both numeric functions, such as recursive functions, and non-numeric functions, such as artificial intelligence problems. This flexibility is due to the use of LISP as a base language. Within a function, several LISP operations can be performed in parallel (such as binding lookup, preparation to call, return from call). In certain cases, such as MAP functions, we can also perform large numbers of function invocations in parallel, provided enough physical processors are available.

This work also confirms the findings of Warren, Ellis, and Fisher concerning the effectiveness of multiple stack processors [Warren 1983] and wide instruction words [Ellis 1986], [Fisher 1981], [Fisher 1984], [Fisher 1987].

11.1. Specific Goals Achieved

A uniprocessor simulation at the module level has been developed. This module level simulator is capable of executing the full single-processor instruction set (within the memory limits of the simulation host hardware). This simulator is written in LISP.

System support software, such as a LISP compiler, assembly-language optimizer, assembler, and loader have been developed and tested. This software is also written in LISP and supports a useful subset of Common LISP.

The results of the benchmark suite indicate that the performance of single processor DIS machine, constructed of commercially available components, to be on the order of ten to fifteen times that of other architectures designed for LISP, and on the order of twice or more the speed of a Cray-1 running LISP. A great portion of this speed increase is due to the multiple internal paths which can be used to execute various parts of a function in parallel. The compiler and optimizer take advantage of these parallel paths.

11.1.1. What Features of DIS Cause the Speedup?

The single most important factor the DIS machine's performance is the high internal bandwidth of the DIS architecture. This bandwidth is used at several levels, both by the model the compiler uses for the system, and by the optimizer. Without the high bandwidth and flexibility of the crossbar, there would be little the compiler or the optimizer could do to make LISP run more quickly.

To cite examples, one obvious usage of the high bandwidth is the use of multiple memories. For example, M0 always contains instructions, but because nothing *but* instructions is stored in M0, we never have to wait for an instruction because data motion needs the memory. This avoids the so-called "von Neumann bottleneck" (having only a single data path from CPU to memory).

Another example of the use of the flexible high bandwidth crossbar is the way that function call and return are overlapped with argument stacking/result calculation. By prestoring a call frame on the pending stack, we can overlap the creation of function arguments with the actual calling. Likewise, by having the return frame information directly available on the return stack, we can overlap calculation of the function's result with the actual return.

Splitting the symbol table memory (M1) from consing memory (M2) gives a similar speedup, because we can overlap the fetches of global data value pointers. We can also overlap the accessing of named entry points (functions named in the symbol table) with pointer-following in the consing memory area. We also can pipeline usage of the binding stack, by the same means (first pass an address both to M1 and the binding stack, pushing the address alone, then subsequent pushes onto the binding stack push both saved values and the next dynamically bound address)

All of this flexibility would be useless without a compiler that can understand and use the flexibility of the interconnection system. Although the standard recursive-descent compiler technology is adequate for initial code generation, it is necessary to use the artificial intelligence techniques of forward and backward chaining to obtain a reasonably optimal machine program.

I would therefore conclude that the combination of the high-bandwidth multiple functional units, the flexibility of the crossbar interconnect, and the intelligence in the compiler to utilize the bandwidth and the flexibility are all essential elements for obtaining the speedups noted. I do not believe that any two of the three elements would have been effective without the third.

11.2. Hardware Implementation Issues

11.2.1. Crossbar

As detailed in section 7.2, several options exist concerning actual construction of the DIS machine. In a uniprocessor prototype, it seems most reasonable to go with the simplest and cheapest (and most restrictive) configuration; the restricted (32-bit) crossbar. This will provide an operational machine for testing more surely and inexpensively than the other options listed.

11.2.2. Memory Arrays

The prototype restricted-crossbar DIS machine could have ten 32-bit memory arrays (for M1, M2, and two per Stack Unit), and one 256-bit array (for M0). As it may be most convenient to build (or buy) only 32-bit arrays, a total of 18 32-bit arrays are needed. If we use currently (1987) available parts, such as 64K by 4 bit wide (256K bits/chip) 45 nSec RAMs, we will have a minimum system configuration with about 2.3 megabytes of memory. This is smaller than we might want, so an ability to expand memory arrays is warranted. Going to a 256Kx1 RAM chip in the arrays would put 9.2 megabytes into the smallest configuration.

Because some units need more memory than others (it is reasonable to assume that there will be more conses than symbol table entries), it may be worthwhile to design the memory array cards to accept either a 64Kx4 part or a 256Kx1 part. In this way, extra memory can be added to M2 (cons cell memory) without adding it to M1 (symbol table memory)

11.2.3. Memory Indexing

Many of the adders in the memory units are used to add an offset of zero through seven. Because the compiler can be forced to generate only aligned references to these addresses, it may be possible to replace these adders with OR gates affecting only the low order bits of the address. This would be less expensive, than a full adder, as well as speeding up the memory cycle time.

It should be noted that we *do* need to keep a true addition capability in the adders for the Stack Units, because in general accesses on the Stack Units are *not* aligned to even address boundaries.

11.2.4. Cache

It would probably be unwise to build any cache units on the first prototype. Until reasonably large problems can actually be run (not simulated), it is difficult to determine the best cache sizes and cache use strategies.

If the design of the memory arrays is properly done, it should be possible to allow a cache unit to be interposed between any of the memory or stack controller cards and the actual memory arrays, once testing shows that cache would be a useful addition.

11.3. Possible Defects in This Research

11.3.1. Benchmark set too small

This size of the benchmark set was forced by the need for a bit-level simulator, and the consequent slowness of that simulation. If the simulator were written at a higher level, it would have run more quickly, but the accuracy of the simulation itself would have been called into doubt.

The author finds it preferable to present a small set of results, with a very high confidence in the results, than a large set of results with a low confidence.

11.3.2. Compiler "Caters" to the Benchmarks

Because the compiler is a recursive-descent system, it does not do any recognition of the particulars of an expression beyond the special forms and the inline forms (arithmetic, `car`, `cdr` and `eq`). All other forms are handled by the same algorithm (build call frame, call function via symbol table address).

It is therefore unlikely that the compiler in any way caters to produce good code on the particular benchmarks given.

11.3.3. DIS Does Not Do Interrupts

The DIS machine was not designed to operate standalone. Instead, a host processor was intended to control the DIS machine via the boot/debug processor or network interface. This host processor (interconnected to the DIS machine via a parallel interface) would be responsible for fielding interrupts, performing I/O, and other user-interfacing tasks.

The use we envision for a DIS machine is as a "back-end" processor, executing a large LISP program, while the host processor periodically scans out result areas and interacts with the user. User input is buffered until the DIS LISP program requests further input from the host processor. In this context as a back-end machine, lack of real-time interrupt capability is not disabling.

11.4. Hindsight Comments

If the DIS project were to be run again, there would be relatively few changes I would make, given the personnel and financial constraints of the project.

If finances allowed, it would have been worthwhile to construct a 1/10 speed prototype. This 1/10 speed prototype could use a single multiplexed bus, rather than the crossbar. This prototype would have allowed us to run a much larger set of benchmarks, and experiment more with various optimization rules in the rule-based optimizer. By itself, this 1/10 speed prototype would have been as fast as any commercially available LISP machine.

If possible, I would have avoided the use of diskless workstations. The DIS team worked for about six months on several diskless workstations and the team consensus was that a LISP on a small-memory diskless workstation is about as useful as a screen door on a submarine. I believe the final success of the DIS team was as much a result of the loan of a fully-equipped AI VAXstation (courtesy Digital Equipment Corporation) as any other single factor.

11.5. Directions for Further Research

The DIS architecture represents a significant advance in the architecture of computer systems. The existence proof of the inherent parallelism in LISP generated by the DIS hardware and software opens several questions in the field of computer design and artificial intelligence.

11.5.1. How Well Does DIS Utilize Multiple CPU's?

This question is currently under exploration by Randall Shane. We expect that the answer is "quite well", and our hand analysis shows that it ought to work. However, the real proof will be the demonstration of a multiprocessor-utilizing DIS compiler, and until that happens, the question must be treated as open.

11.5.2. Does DIS Work Well with other Languages?

The DIS compiler compiles only Common LISP directly from LISP expressions. The effectiveness of DIS on languages such as FORTRAN or C has not been demonstrated. The implication of the VLIW work is that the wide instruction word aspects of DIS would be useful [Ellis 1986], but the replacement of the VLIW register banks with DIS stack units makes this an open question.

Because the DIS data path is a superset of the Warren Abstract PROLOG Architecture, it is likely that a DIS machine could run PROLOG very well. We haven't written a PROLOG compiler for the DIS machine, so we don't know this for sure [Despain 1985], [Dobry 1985], [Warren 1983].

11.5.3. Will a Wider DIS Machine go faster?

Currently the point of greatest contention in the DIS machine is the Value Stack. A DIS machine with multiple Value Stacks might be faster, if the compiler and optimizer can find a way to utilize the extra value stacks effectively.

Multiflow Computer, Inc. indicates that their VLIW machine can be field-expanded to a wider instruction word and more functional units, and their FORTRAN compiler is capable of utilizing the increased number of functional units.

11.5.4. Can DIS run non-procedural languages?

Clearly DIS can run LISP interpreters of the common non-procedural languages such as PROLOG and OPS5. Because the DIS datapath and user-visible storage is a superset of the Warren PROLOG machine datapath, it is likely that DIS could run PROLOG well. The question of OPS5 is an open question. Although the original OPS5 systems were LISP-based, several native-mode OPS5 compilers are now running, and these compilers do not follow the LISP interpretation model [Barabash 1986].

References

- [Agrawal 1983] Agrawal, D. P., "Graph Theoretical Analysis and Design of Multistage Interconnection Networks", IEEE Trans. on Computers, Vol. C-32, No. 7, July 1983
- [Agrawala 1983] Agrawala, A., Herzog, U., "Guest Editors Introduction: Performance Evaluation of Multiple Processor Systems", IEEE Trans. on Computers, Vol C-32, No. 1, January 1983
- [Alighieri 1315] Alighieri, D., "The Inferno", Vol. 1 of The Divine Comedy ca. 1315 AD.
- [Baer 1983] Baer, J-L., Du, H-C., "Binary Search in a Multiprocessing Environment, IEEE Trans. on Computers", Vol C-32, No. 7, July 1983
- [Barabash 1986] Barabash, B., private communications on the OPS5 RETE algorithm, May 1986 to August 1986
- [Batcher 1980] Batcher, K., "Design of a Massively Parallel Processor", IEEE Trans. on Computers, Vol C-29, No. 8, September 1980, pp. 836-840
- [Batcher 1982] Batcher, K., "Bit-Serial Parallel Processing Systems", IEEE Trans. on Computers, Vol C-31, No. 5, May 1982
- [Bawden 1984] Bawden, A., Agre, P., "What good are a million processors if they all do the same thing?", draft of paper submitted to AAAI 1984, MIT AI laboratory, 545 Technology Square, Cambridge, MA
- [Bhuyan 1983] Bhuyan, L., Agrawal, D., "Design and Performance of Generalized Interconnection Networks", IEEE Trans. on Computers, Vol. C-32, No. 12, December 1983
- [Bhuyan 1984] Bhuyan, L., Agrawal, D., "Generalized Hypercube and Hyperbus Structures for a Computer Network", IEEE Trans. on Computers, Vol C-33, No. 4, April 1984
- [Bisiani 1977] Bisiani, R., "Paging Behavior of Knowledge Networks, Report CMU-CS-77-160, Carnegie-Mellon University, Department of Computer Science. August 1977
- [Boari 1984] Boari, M., Crespi-Reghizzi, S., Dapra, A., Maderna, F., Natali, A., "Multiple-microprocessor programming techniques: MML, a New Set of Tools", IEEE Computer, January 1984
- [Buehrer 1982] Buehrer, R., Brundiens, H-J., Bron, B., Friess, H., Haelg, W., Halin, H., Isacson, A., Tadian, M., "The ETH-Multiprocessor EMPRESS: A Dynamically

Configurable MIMD System", IEEE Trans. on Computers, Vol C-31, No 11, Nov 1982

[Byte 1981] Byte, "Smalltalk Special Issue", Byte, August 1981 (entire issue)

[Chou 1983] Chou, T., Abraham, J., "Load Redistribution Under Failure in Distributed Systems", IEEE Trans. on Computers, Vol. C-32, No. 9, September 1983

[Christ 1984] Christ, N., Terrano, A., "A Very Fast Parallel Processor", IEEE Trans. on Computers, Vol. C-33, No. 4, April 1984

[Chu 1981] Chu, Y., Abrams, M., "Programming Languages and Direct-Execution Computer Architecture", IEEE Computer, July 1981

[Chu 1984] Chu, W., Lan, M., Hellerstein, J., "Estimation of Intermodule Communication (IMC) and it's Applications in Distributed Processing Systems", IEEE Trans. on Computers, Vol C-33, No. 8, Aug 1984

[Clark 1981] Clark, D., Lampson, B., Pier, K., "The Memory System of a High-Performance Personal Computer", Xerox PARC, 3333 Coyote Road, Palo Alto, CA 94304, January 1981

[Davis 1980] Davis, C., "Ballistic Missile Defense: A Supercomputer Challenge", IEEE Computer, November 1980

[Dennis 1984] Dennis, J., Gao, G., Todd, K., "Modeling the Weather with a Data Flow Supercomputer", IEEE Trans. on Computers, Vol C-33, No 7, July 1984

[Despain 1985] Despain, A. "A High-Performance PROLOG Co-Processor", Proceedings of Wescon 85, September 1985.

[Digital 1979] Digital Equipment Corporationnb "VAX Architecture Handbook", 1979

[Dobry 1985] Dobry, T. P., Despain, A. M., Patt, Y. N., "Performance Studies of a Prolog Machine Architecture", Proceedings of 12'th International Symposium on Computer Architecture, June 1985

[Doran 1979] Doran, R., "Computer Architecture: A Structured Approach", Academic Press, 1979

[Edelson 1984] Edelson, B., "Computational Investigations Utilizing the Massively Parallel Processor (MPP)", Space Science and Applications Notice, 20 December 1984, National Aeronautics and Space Administration, Washington DC 20546

[Ellis 1986] Ellis, John R., "BULLDOG: a Compiler for VLIW Architectures", ACM Doctoral Dissertation Award Winner for 1985, MIT Press, 1986,

- [Fathi 1983] Fathi, E., Krieger, M., "Multiple Microprocessor Systems: What, Why, and When", IEEE Computer, March 1983
- [Fisher 1981] Fisher, J. A., "Trace Scheduling: a Technique for Global Microcode Compaction", IEEE Trans. on Computers, Vol C-30, No. 7, July 1981
- [Fisher 1984] Fisher, J. A., "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", IEEE Computer, July 1984
- [Fisher 1987] Fisher, J. A., "Wide Instruction Word Architectures: Solving the Supercomputer Software Problem", paper submitted to INRIA International Seminar on Scientific Supercomputers, February 2-6, 1987, Paris, France
- [Foderaro 1981] Foderaro, J., Sklower, K., "The Franz LISP Manual", University of California at Berkeley, 1981
- [Foster 1972] Foster, C. C., Riseman, E. M., "Percolation of Code to Enhance Parallel Dispatching and Execution", IEEE Trans. on Computers, December 1972
- [Fuller 1978] Fuller, S., Osterhout, L., Rubinfeld, P., Sindhu, P., Swan, R., "Multi-Microprocessors: An Overview and Working Example", Proc. IEEE, Vol 66, No. 2, February 1978, pp. 216-227
- [Gabriel 1984] Gabriel, R. P., "Performance and Evaluation of Lisp Systems" Final Report, Stanford Lisp Performance Study, Computer Science Department, Stanford University, October 1984
- [Garcia-Molina 1984] Garcia-Molina, H., Lipton, R., Valdes, J., "A Massive Memory Machine", IEEE Trans. on Computers, Vol C-33, No. 5, May 1984
- [Gonzalez 1972] Gonzalez, M., Jr., Ramamoorthy, C., "Parallel Task Execution In a Decentralized System", IEEE Trans. on Computers, Vol C-21, No. 12, December 1972, pp. 1310-1322
- [Gostelow 1980] Gostelow, K., Thomas, R., "Performance of a Simulated Dataflow Computer", IEEE Trans. on Computers, Vol C-29, No. 10, October 1980
- [Gottlieb 1983] Gottlieb, A., Grishman, R., Krustal, C., McAuliffe, K., Rudolph, L., Snir, M., "The NYU Ultracomputer- Designing an MIMD Shared Memory Parallel Computer", IEEE Trans. on Computers, Vol C-32, No. 2, Feb 1983
- [Guzman 1981] Guzman, A., "A Heterarchical Multi-Microprocessor LISP Machine", Technical Report AHR-81-17, University of Mexico, also presented 1981 IEEE Computer Workshop on Computer Architecture for Pattern Analysis and Image Database Management.
- [Hack 1986] Hack, J. J., "Peak vs. Sustained Performance in Highly Concurrent Vector Machines", IEEE Computer 19(9), September 1986

- [**Hillis 1984**] Hillis, D., "Fine-Grained machines", Private communication 7/14/1984
- [**Hillis 1985**] Hillis, D., "The Connection Machine", MIT Press, Cambridge MA., 1985
- [**Hodges 1983**] Hodges, A., "Alan Turing: the Enigma", Simon and Shuster, 1983
- [**Hoffman -**] Hoffman, R., "User's Manual for GLYPLIT", Rand Corp, Santa Monica, California, available NTIS DAHC15-67-C-0141 (3x5 microfiche)
- [**Kartashev 1982**] Kartashev, S. P., "Supersystems: Current State-of-the-Art: Guest Editors Introduction", IEEE Trans. on Computers, Vol C-31, No. 5, May 1982
- [**Keller 1979**] Keller, M. K., Lindstrom, G., Patil, S., "A loosely coupled applicative multi-processing system", Proc. 1979 National Computer Conference, pgs 613-622, Vol. 48, AFIPS.
- [**Knight 1984**] Knight, T., Poggio, T., "Connection Machine Architecture and Vision Algorithms", draft of preliminary presentation to DARPA Strategic Computing project, MIT AI Laboratory, 545 Technology Square, Cambridge, MA
- [**Kuck 1982**] Kuck, D., "The Burroughs Scientific Processor", IEEE Trans. on Computers, Vol C-31, No. 5, May 1982
- [**Kushner 1982**] Kushner, T., Wu, A., Rosenfeld, A., "Image Processing on ZMOB", IEEE Trans. on Computers, Vol C-31, No. 10, October 1982
- [**Lampson 1981a**] Lampson, B., Pier, K., "A Processor for a High-Performance Personal Computer", Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304, January 1981
- [**Lampson 1981b**] Lampson, B., McDaniel, G., Ornstein, S., "An Instruction Fetch Unit for a High-Performance Personal Computer", Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304, January 1981
- [**Lincoln 1982**] Lincoln, N., "Technology and Design Tradeoffs in the Creation of a Modern Supercomputer", IEEE Trans. on Computers, Vol C-31, No. 5, May 1982
- [**Lincoln 1983**] Lincoln, N., "Supercomputers = Colossal Computations + Enormous Expectations + Renowned Risk", IEEE Computer, May, 1983
- [**Manner 1984**] Manner, R., "Hardware Task/Processor Scheduling in a Polyprocessor Environment", IEEE Trans. on Computers, Vol C-33, No 7, July 1984
- [**Marsan 1983**] Marsan, M., Balbo, G., Conte, G., Gregoretti, F., "Modeling Bus Contention and Memory Interference in a Multiprocessor System", IEEE Trans. on Computers, Vol. C-32, No. 1, January 1983

- [Moko-ota 1984] Moko-ota, T., Stone, S., "Fifth-Generation Computer Systems: A Japanese Project", IEEE Computer, March 1984
- [Moon 1985] Moon, D. A., "Architecture of the Symbolics 3600", IEEE Symposium on Computer Architecture, June 1985. IEEE order number 0149-7111/85/0000/0076 \$0100
- [Moon 1987] Moon, D. A., "Symbolics Architecture", IEEE Computer, Vol. 20, No. 1, January 1987
- [Nilsson 1984] Nilsson, M., "The World's Shortest Prolog Interpreter?", Implementations of Prolog, edited by J.A. Campbell, Halstead press, 1984.
- [Norrie 1984] Norrie, C., "Supercomputers for Superproblems: An Architectural Introduction", IEEE Computer, March 1984
- [Oruc 1984a] Oruc, A., "A Classification of Cube-Connected Networks with a Simple Control Scheme", IEEE Trans. on Computers, Vol C-33, No. 8, August 1984
- [Oruc 1984b] Oruc, A., Prakash, D., "Routing Algorithms for Cellular Interconnection Arrays", IEEE Trans. on Computers, Vol C-33, No. 10, October 1984
- [Papert 1980] Papert, S., "Mindstorms", Basic Books, Inc., 1980
- [Ponder 1983] Ponder, C., "...but will RISC run LISP?? (a feasibility study)", Univ. of California, Dept. of Electrical Engineering and Computer Science, Computer Science Division, May 11, 1983
- [Riseman 1972] Riseman, E. M., Foster, C. C., "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Trans. on Computers, December 1972.
- [Reed 1983] Reed, D., Schwetman, H., "Cost-Performance Bounds for Multimicrocomputer Networks", IEEE Trans. on Computers, Vol. C-32, No. 1, January 1983
- [Rich 1984] Rich, E., "The Gradual Expansion of Artificial intelligence", IEEE Computer, May 1984
- [Rumbaugh 1977] Rumbaugh, J., "A Data-Flow Multiprocessor", IEEE Trans. on Computers, Vol C-26, No. 2, February 1977
- [Schwartz 1983a] Schwartz, J., "Design Alternatives for Ultraperformance Parallel Computers", paper given at NSF Information Technology Workshop, October 1, 1983.
- [Schwartz 1983b] Schwartz, J., "A Taxonomic Table of Parallel Computers, Based on 55 Designs", Ultracomputer Note #69, November 1983
- [Seban 1984] Seban, R., Siegel, H., "Shuffling with the Illiac and PM2I SIMD Networks", IEEE Trans. on Computers, Vol C-33, No 7, July 1984

- [Shane 1987] Shane, R. H., series of private communications and discussions, 1986 through 1987.
- [Shin 1982] Shin, K., Lee, Y-H., Sasidhar, J., "Design of HM2p- A Hierarchical Multimicroprocessor for General-Purpose Applications", IEEE Trans. on Computers, Vol C-31, No. 11, November 1982
- [Siegel 1981a] Siegel, H., McMillen, R., "Using the Augmented Data Manipulator Network in PASM", IEEE Computer, February 1981
- [Siegel 1981b] Siegel, H., Siegel, L., Kemmerer, F., Mueller, P., Smalley, H., Smith, D., "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition", IEEE Trans. on Computers, Vol C-30, No. 12, December 1981
- [Steele 1981] Steele, G., Sussman G., "Scheme-79: LISP on a Chip", IEEE Computer Magazine, pgs 10-21, July 1981, pp. 10-21
- [Steele 1982] Steele, G., "Common LISP Reference Manual", Carnegie Mellon University, Pittsburgh, PA 15213, July 1982
- [Stefik 1982] Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., Sacerdoti, E., "The Organization of Expert Systems, A Tutorial", Artificial intelligence 18 (1982)
- [Stone 1984] Stone, H., "Database Applications of the Fetch-And-Add Instruction", IEEE Trans. on Computers, Vol C-33, No 7, July 1984
- [Stout 1983] Stout, Q., "Mesh-Connected Computers with Broadcasting", IEEE Trans. on Computers, Vol C-32, No. 9, September 1983
- [Tick 1983] Tick, E., Warren, D. H., "Towards a Pipelined Prolog Processor", Tech Report, AI Center, SRI International, Menlo Park, CA (Aug 1983)
- [Tjaden 1970] Tjaden, G. S., Flynn, M. J., "Detection and Parallel Execution of Independent Instructions", IEEE Trans. on Computers, Vol C-19, No. 10, October 1970
- [Torng 1984] Torng, H.C., "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", Technical Report EE-CEG-84-7, Cornell University, Electrical Engineering Department.
- [Treleaven 1984] Treleaven, P., Lima, I., "Future Computers: Logic, Data Flow,...., Control Flow?", IEEE Computer, March 1984
- [Verac -] Verac, Inc, "OPS5e User's Manual", (Symbolics 3600 version)
- [Vick 1980] Vick, C., Kartashev, S. P., Kartashev, S. I., "Adaptable Architectures for Supersystems", IEEE Computer, November 1980

- [VonConta 1983] Von Conta, C., "Torus and Other Networks as Communication Networks with up to Some Hundred Points", IEEE Trans. on Computers, Vol. C-32, No. 7, July 1983
- [Waltx 1987] Waltx, D. L., "Applications of the Connection Machine", IEEE Computer, Vol. 20, No. 1, January 1987
- [Warren 1983] Warren, D. H., "An Abstract Prolog Instruction Set", Technical Note 309, AI Center, SRI International, Menlo Park, CA 94025 (1983)
- [Weinreb 1981] Weinreb, D., Moon, D., "LISP Machine Manual", reprinted by Symbolics, Inc., under license from the MIT Artificial intelligence Laboratory.
- [Wittie 1980] Wittie, L., Van Tilborg, A., "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer", IEEE Trans. on Computers, Vol. C-29, No. 12, December 1980
- [Yamamoto 1981] Yamamoto, M., "A Survey of High-Level Language Machines in Japan", IEEE Computer, July 1981
- [Zakharov 1984] Zakharov, V. "Parallelism and Array Processing", IEEE Transactions on Computers, Vol C-33, No. 1, January 1984

Index

AHR :	17
AI VAXstation :	91
ALU :	48
AMPS :	17
APL :	25
Agrawal 1983 :	35
Agrawala 1983 :	5
Aligheiri 1315 :	40
Applicative Architecture :	7
Applicative programming :	22
Applicativeness, computability :	26
Artificial intelligence :	2
Associative memory :	10
BSP :	11
Back-door bus :	41
Banyan :	14
Barabash 1986 :	105
Basic block :	67
Batcher 1980 :	9
Batcher 1982 :	10
Bawden 1984 :	11
Benchmarks :	67, 94
Bhuyan 1983 :	35
Bhuyan 1984 :	35

Binding stack :	48
Bombe :	8
Boot/debug processor :	53
Branch conflict :	68
Buehrer 1982 :	13
Byte 1981 :	15
C :	104
C.mmp :	12
CADR :	16
Cache :	102
Cache :	7
Christ 1984 :	10
Chu 1981 :	6
Chu 1984 :	35
Clark 1981 :	15
Cm* :	12
Code generator stubs :	65
Cold-boot :	72
Compiler :	63
Condition codes :	49, 51, 71
Conditional branch :	58
Conditional call :	58
Conflict-based optimization :	68
Connection Machine :	10, 11
Control Data Corp. :	4

Cray X-MP :	11
Cray-1 :	4, 9, 11, 91
Crossbar :	101
Crossbar construction :	55
Cyber 205 :	11
DIS architecture :	39
Dataflow :	24
Dataflow conflict :	68
Deep binding :	30
Dennis 1984 :	24
Despain 1985 :	104
Despain 1985 :	18
Distribution bus :	54
Dobry 1985 :	18, 104
Doran 1979 :	4
EMPRESS :	13
Edelson 1984 :	9
Ellis 1986 :	18, 104
Enigma :	8
Evlis :	16
Explorer :	91
FGL :	17
FORTRAN :	25, 104
Factorial :	83, 96
Fetch and add :	14

Fiber :	56
Fisher 1981 :	18
Fisher 1984 :	18
Fisher 1987 :	18
Flip unit :	10
Foderaro 1981 :	28
Foster 1972 :	19
Fuller 1978 :	5, 12
GO :	41
Gabriel 1984 :	91
Gabriel :	94
Garbage collection :	59
Garcia-Molina 1984 :	41
Gonzalez 1972 :	35
Gostelow 1980 :	24
Gottlieb 1983 :	14
Guzman 1981 :	17
Hack 1986 :	5
Hillis 1985 :	10
Hillis 1985 :	11, 60
Hodges 1983 :	8, 40
Hoffman :	2
Hoffman :	9
Host processor :	53, 103
ILLIAC IV :	9

Inline forms :	64
Interrupts :	103
Keller, 1979 :	17
Knight 1984 :	11
Kuck 1982 :	11, 57
Kung, H. T. :	10
Kushner 1982 :	13
LISP 1.5 :	37
LISP :	1, 25, 28
LISP argument passing :	29
LISP machines :	15
LISP variable scoping :	29
LMI :	15
Lampson 1981a :	6, 15
Lampson 1981b :	15
Lincoln 1982 :	11
Linker :	70
Lisp Machine Inc. :	15, 16
M0 :	43
M1 :	43
M2 :	43
MEF :	17
MIMD :	6
MODULE-WAIT :	41
MPP :	9

Manner 1984 :	13
Mark and sweep :	60
Marsan 1983 :	35
Memory :	42, 102
Microcode :	15
Microwave :	56
Moon 1987 :	16
Multiflow Computer, Inc. :	18, 105
Multiprocessor :	5
NREV :	96
Network interface :	52
Nilsson 1984 :	18
Norrie 1984 :	4
OPS5 :	105
Optimizer :	66
Oruc 1984a, 1984b :	35
PDP-11 :	12
PROLOG :	104
PSL :	92
Paracomputer :	14
Parentheses :	28
Pending stack :	45
Pipeline :	61
Ponder 1983 :	36
Prolog :	18

RDIV :	94
Recursive-descent :	63
Reservation conflict :	68
Resource conflict :	68
Return stack :	47
Riseman 1972 :	19
Rule-based conflict resolution :	68
Rule-based merger :	69
Rumbaugh 1977 :	24
SIMD :	6, 8
SISD :	6
STAK :	94
STARAN :	10
Scheme-79 :	16
Schwartz 1983a :	14
Schwartz 1983b :	8
Seban 1984 :	9
Sequence conflict :	68
Sequencer :	50
Serialization :	14
Shallow binding :	30
Shane 1987 :	43
Simulator :	70
Single-assignment programming :	24
Special declaration :	30

Stack :	7, 23, 43
Steele 1981 :	16
Steele 1984 :	28
Stone 1984 :	14
Symbol tables :	65
Symbolics :	15, 16, 91
System definition macros :	63
Systolic arrays :	11
TAK :	94
TAKL :	94
Tag bits :	15, 59
Texas Instruments :	15, 16, 91
Thinking Machines Inc. :	11
Tick 1983 :	18
Tjaden 1970 :	2, 19
Torng 1984 :	4
Ultracomputer. :	14
VAX Common LISP :	92
VLIW :	18, 104
Value stack :	45, 105
Vector processors :	4, 11
Verac :	16
Virtualization :	7
Von Conta 1983 :	35
Von Neumann Architecture :	7

Von Neumann bottleneck :	101
Waltx 1987 :	11
Warren 1983 :	104
Warren 1983 :	18
Waveguide :	56
Weinreb 1981 :	16
Weinreb 1981 :	6
Weinreb 1981 :	6
Xapping :	11
Xectors :	11
Xenologic, Inc :	18
Yamamoto 1981 :	16
ZMOB :	13
Zakharov 1984 :	10
ZetaLISP :	92

Appendix A

DIS File Listings


```

;   Base simulator functions
;   (this file should be compiled

(defun power (x n)
  (cond
    ((equal x 2)
     (powertable x n))
    (t (powerhard x n))))

(defun powerhard (x n)
  (cond
    ((equal n 0) 1)
    (t (* x (power x (- n 1))))))

(setq powtabl (make-array 512))

(defun powertable (x n)
  (cond
    ((null (aref powtabl n))
     (setf (aref powtabl n) (powerhard x n)))
    (t (aref powtabl n))))

(power 2 50)
(power 2 100)
(power 2 150)
(power 2 200)

;
;   Extract - value is the bits in,
;   rangelist is '(bits-wide starting-at-bit)
;   we count bits with the rightmost being bit zero.
(defun extract (value rangelist)
  (-
    (ash value (- (cadr rangelist))) ;get rid of low order waste bits
    (ash
      (ash value (- (+ (car rangelist) (cadr rangelist))))
      (car rangelist))))

```



```

;
; define the s- selector function
(putd
  (implode (append (explode 's-) (explode fieldname)))
  (list 'lambda '())
  (list 'extract 'driven-instruction
    (list 'quote
      (list (eval width) (eval startloc))))))
;
; account for the bits taken up-
(setq startloc (+ startloc (eval width)))
)
;
;
;
; And now the fields are defined:
;
; M0 - usually used for instruction fetch
;
(deffield m0v0 4) ;M0 address mux input
(deffield m0v1 4) ;M0 Data mux input
(deffield m0v2 3) ;M0 offset (0-7 words added to address)
(deffield m0c 2) ;M0 control - idle/read/write/fetchandadd
;
; M1 - usually used for conses
;
(deffield m1v0 4) ;M1 address mux input
(deffield m1v1 4) ;M1 Data mux input
(deffield m1v2 3) ;M1 offset (0-7 words added to address)
(deffield m1c 2) ;M1 control - idle/read/write/fetchandadd
;
; M2 - usually used for other things
;
(deffield m2v0 4) ; M2 address input (usually a seq #)
(deffield m2v1 4) ; M2 data input (usually a value)
(deffield m2v2 3) ; M2 offset (0-7 words added to addr)
(deffield m2c 2) ; M2 control - idle/read/write
;
; Value stack
;
(deffield vv0 4) ;Value stack first mux input
(deffield vv1 4) ;Value stack second mux input
(deffield vc0 2) ;Value stack subhead read (use vv1 for stackpointer)
(deffield vc1 4) ;Value stack stackpointer motion control (- 8)
(deffield vc2 2) ;Value stack write actional (after pointer motion)
(deffield vc3 1) ;Value stack write resultant into VSP
;
; Pending stack
;
(deffield pv0 4) ;pending stack first mux input
(deffield pv1 4) ;pending stack second mux input
(deffield pc0 2) ;pending stack subhead read (use pv1 for stackpointer)
(deffield pc1 8) ;pending stack stackpointer motion control (- 8)
(deffield pc2 2) ;pending stack write actional (after pointer motion)
(deffield pc3 1) ;pending stack write resultant into PSP
;
; Return stack
;
(deffield rv0 4) ;Return stack first mux input
(deffield rv1 4) ;Return stack second mux input
(deffield rc0 2) ;Return stack subhead read (use rv1 for stackpointer)
(deffield rc1 8) ;Return stack stackpointer motion control (- 8)
(deffield rc2 2) ;Return stack write actional (after pointer motion)

```

```

(deffield rc3 1)      ;Return stack write resultant into RSP
;
;   Binding stack
(deffield bv0 4)      ;Binding stack first mux input
(deffield bv1 4)      ;Binding stack second mux input
(deffield bc0 2)      ;Binding stack subhead read (use bv1 for stackpointer)
(deffield bc1 4)      ;Binding stack stackpointer motion control (- 8)
(deffield bc2 2)      ;Binding stack write actional (after pointer motion)
(deffield bc3 1)      ;Binding stack write resultant into BSP
;
;   ALU
(deffield av0 4)      ;ALU first mux input
(deffield av1 4)      ;ALU second mux input
(deffield ac0 4)      ;ALU boolean operation
(deffield ac1 2)      ;ALU data selector (0, 1, -1, av0 )
(deffield ac2 1)      ;ALU format operation selector (fix/float/comp)
(deffield ac3 2)      ;ALU operation ( add, -, *, /)
(deffield ac4 8)      ;ALU shift count
;
;   Network Interface
(deffield nv0 4)      ;Network Interface mux input (usually data director)
(deffield nc0 2)      ;network operation (offer/accept/idle)
;
;   Sequencer
(deffield sv0 4)      ;Sequencer next instruction in
(deffield sv1 4)      ;Sequencer 'normal' next instr. address
(deffield sv2 4)      ;Sequencer 'branch' next instr. address
(deffield sc0 2)      ;Sequencer add 0, 1, 2, 3 to "normal" next addr
(deffield sc1 2)      ;Sequencer add 0, 1, 2, 3 to "branch" next addr
(deffield sc2 5)      ;Sequencer condition code mask
; (kanbanzero, kanbannegative, kanbannil
; taskin, taskout)
(deffield sc3 3)      ;Sequencer conditional wait mask- ignore module ready?
;
;   CMI - actually, part of the sequencer, but it's a mux...
(deffield cc0 3)      ; Selection of CMI (imm. data, or sp's or
; whatever.
(deffield cv0 32)      ; CMI (immediate data) . It has two names,
; CMI and cv0. Watch out.
(deffield cv1 16)      ; "pointer-count" field of CMI
(deffield cv2 16)      ; "element-count" field of CMI
; Remember, pointers > elements is
; meaningless, so p=1,e=0 is defined as NIL
;
;
(princ "bits used :")
(princ (+ startloc 1))
(terpri)
(setq bd-field-un bd-field)
;
;   macro to help define busnames
(defmacro defbus (busname)
  (cond ((not (null print-commentary))
    (princ busname) (princ " is bus number ")
    (princ (eval buscount)) (terpri)))
  (set busname (eval buscount))
  (setq buscount (+ buscount 1))
  )
)
(defmacro buscountzero ()
  (setq buscount 0))
(terpri)
(buscountzero)
;

```

```

;       define the busses themselves
(defbus b-md0)      ; M0 data out
(defbus b-md1)      ; M1 data out
(defbus b-md2)      ; M2 data out
(defbus b-vd0)      ; top of value stack
(defbus b-vd1)      ; one below top of value stack
(defbus b-vsp)      ; value stack pointer
(defbus b-pd0)      ; pending call top location (usually code)
(defbus b-pd1)      ; pending call top-1 (usually argument pointer)
(defbus b-rd0)      ; return stack top (usually return location)
(defbus b-rd1)      ; return stack top-1 (usually argument return pointer)
(defbus b-bd0)      ; binding stack top (usually old value)
(defbus b-bd1)      ; binding stack top-1 (usually seq # and/or address)
(defbus b-ad0)      ; ALU output 0 (low order bits)
(defbus b-sd0)      ; sequencer next address (usually goes to m0 read)
(defbus b-nd0)      ; network output bus
(defbus b-cmi)      ; CoMmand Immediate bus (also may have sptrs)

;
;       Now we create the memory of the system- because the back
;       door bus exists, we can consider all of memory to be one long
;       array (for the uniprocessor, at least).
;
;       True to form, we define a macro to help
(princ "Creating the memory spaces ... ") (terpri)
(defmacro defspace (spacename length)
  (cond ((not (null print-commentary))
    (princ spacename) (princ " starts at ") (princ startloc)
    (princ " and is of length ") (princ (eval length)) (terpri)))
  (set spacename (cons startloc (eval length)))
  (setq startloc (+ startloc (eval length))))
)
(terpri)
(startloczero)
;
;       and the memory spaces...
(setq memory-module-size 1000)
;
(defspace m0 memory-module-size)
(defspace m1 memory-module-size)
(defspace m2 memory-module-size)
(defspace v0 memory-module-size)
(defspace v1 memory-module-size)
(defspace p0 memory-module-size)
(defspace p1 memory-module-size)
(defspace r0 memory-module-size)
(defspace r1 memory-module-size)
(defspace b0 memory-module-size)
(defspace b1 memory-module-size)
;       now allocate an array;
(setq memory (make-array startloc))
(princ "Memory in simulator: ") (princ startloc) (princ " words.")(terpri)
;
;       define opcodes for various operations - note the commonality!
;       also note that we check to see that there are enough bits for each opcode
;       someday, we could make this even smarter- we look at the opcode fragments
;       and allocate fields when done with definition of opcodes

(princ " Defining op codes ") (terpri)
(defmacro defop (fieldname opname)
  (cond
    ((> (+ 1 startop) (power 2 (car (eval fieldname)))))
    (princ "ERROR! There aren't enough bits in ")
    (princ fieldname)
  ))

```

```

    (princ " to hold that many opcodes ! ") (terpri))
  (t
    (cond ((not (null print-commentary))
      (princ opname) (princ " is defined as decimal ")
      (princ startop) (terpri)))
    (set opname (eval startop)))
; and here to create opcode reversability
;
  (setq bd-field-un (remove (implode (caddr (explode
    fieldname))) bd-field-un))
  (if (not (find (implode (caddr (explode
    fieldname))) bd-field-in))
    (setq bd-field-in (cons (implode (caddr (explode
    fieldname))) bd-field-in)))
  (setq bd-x (implode (append (explode 'bd-)
    (caddr (explode fieldname)))))
  (set bd-x (reverse (eval bd-x)))
  (set bd-x (cons opname (eval bd-x)))
  (set bd-x (reverse (eval bd-x)))
;
  (setq startop (+ startop 1)))
)
)
(defmacro endops()
  (setq startop 0)
)
(terpri)
;
(endops)
; Beware the following Cludge! You *must* have an (endops) after every group
; of opcodes; else they will be consecutive and you will run out of bits!
; If you have an (endops) in the middle of a group of exclusive choices, they
; will cease being exclusive ! (this can be fixed if macros could take
; a variable # of arguments )
;
; WATCH OUT!
;
;
; m0 - idle/read/write/fadd
(defop f-m0c midle) ; do nothing
(defop f-m0c mread) ; read out of memory
(defop f-m0c mwrite) ; write
(defop f-m0c mfadd) ; fetch and add
(endops);
;
; m1 - one field, four choices.
; now uses same choices as M0
;
;
; m2 - one field, three choices.
; now uses same choices as M0
;
; All 4 Stacks - Three control fields
; we define for VS but all 4 stacks take the same operands
;
; field 1 - Subhead read flag
(defop f-vc0 suidle) ; entire stack unit is to idle.
(defop f-vc0 snormal) ; treat SP normally
(defop f-vc0 subhead) ; Use vvl instead of stackpointer
(endops)
;
; field 2 - stackpointer motion control. Sometimes we
; generate these values, sometimes we use the mnemonics.

```

```

;      Watch out.
(defop f-vc1 spop7)
(defop f-vc1 spop6)
(defop f-vc1 spop5)
(defop f-vc1 spop4)
(defop f-vc1 spop3)
(defop f-vc1 spop2)
(defop f-vc1 spop1)
(defop f-vc1 sidle)
(defop f-vc1 spush1)
(defop f-vc1 spush2)
(defop f-vc1 spush3)
(defop f-vc1 spush4)
(defop f-vc1 spush5)
(defop f-vc1 spush6)
(defop f-vc1 spush7)
(defop f-vc1 spush8)
(endops)
;
;      field 3 - write actionals
(defop f-vc2 swidle) ; note vwidle - Write idle !
(defop f-vc2 swtop) ; overwrite top element with VV0
(defop f-vc2 swt-1) ; overwrite t-1 with VV1
(defop f-vc2 swboth) ; overwrite both
(endops)
;
;      field 4 - Do we write the current SP back into the VSP?
(defop f-vc3 spidle) ; Don't write it back
(defop f-vc3 spwrite) ; Write final stptr value (after arithmetic) into VSP
(endops)
;
;      Pending Call Stack - uses same opcodes as value stack
;
;      Return call stack - same opcodes as value stack
;
;      Binding stack hardware - same opcodes as value stack
;
;      ALU - LOTS of fields
;
;      first field- booleans
;      *** DANGER DANGER DANGER *** COMMON LISP DOES NOT DEFINE THE BOOLE
;      FUNCTION CODES IN THE STANDARD WAY. MODIFY THE BELOW AT GREAT PERSONAL
;      DANGER (AND WITH A DEEP UNDERSTANDING OF COMMON LISP *NOT* REFLECTING THE
;      WAY A VAX DOES LOGICAL OPERATIONS)
;
(defop f-ac0 boolzero) ; zero out always ; boole-clr
(defop f-ac0 boolone) ; constant ones always ; boole-set
(defop f-ac0 boola) ; a ; boole-1
(defop f-ac0 boolb) ; b ; boole-2
(defop f-ac0 boolc1) ; c1 ; boole-cl
(defop f-ac0 boolc2) ; c2 ; boole-c2
(defop f-ac0 boolnota) ; ~a ; boole-and
(defop f-ac0 boolnotb) ; ~b ; boole-ior
(defop f-ac0 booland) ; a and b ; boole-xor
(defop f-ac0 boolor) ; a or b ; boole-eqv
(defop f-ac0 boolxor) ; xor ; boole-nand
(defop f-ac0 boolxand) ; ab and ~a~b = ~( a xor b) ; boole-nor
(defop f-ac0 boolnotand) ; ~( a b) ; boole-andc1
(defop f-ac0 boolnotanotb) ; not a not b ; boole-andc2
(defop f-ac0 boolnotab) ; ~a b ; boole-orc1
(defop f-ac0 boolanotb) ; a ~b ; boole-orc2
(defop f-ac0 boolaimpb) ; a implies b
(defop f-ac0 boolbimpa) ; b implies a
(endops)
;
;      ALU data selector
(defop f-ac1 selzero) ; zeroes
(defop f-ac1 sell) ; a one in the LSB

```

```

(defop f-ac1 sel-1)      ; a minus one
(defop f-ac1 selav0)    ; use the value from the "a" (av0) mux.
(endops)
;
;     ALU format selector
(defop f-ac2 fmtfix)    ; use binary arith
(defop f-ac2 fmtfloat)  ; use floating arith
;   (defop f-ac2 fmtcomp) ; do comparisons, not arithmetic, fixed pt
;   (defop f-ac2 fmtfcomp) ; do float comparisons
(endops)
;
;     ALU combinational net
(defop f-ac3 combadd)
(defop f-ac3 combsubtract)
(defop f-ac3 combmultiply)
(defop f-ac3 combddivide)
(endops)
(defop f-ac3 combzero)  ; test- a equal to zero?
(defop f-ac3 combagtb)  ; a greater than b
; we could have two more comparisons if desired?
(endops)
;
;     ALU shifter...rather than describe all 255 different shifts...
(defop f-ac4 shift0)
(defop f-ac4 shift1)
(defop f-ac4 shift2)
(endops)
;
;     Network Interface - one field (control)
(defop f-nc0 nidle)    ; no net task transfer activity
(defop f-nc0 naccept)  ; accept a job on the net
(defop f-nc0 noffer)   ; tells net to take current func. invocation
;                       ; from processor
;                       ; and suggested direction from nv0 address
(endops)
;
;     CMI selection - one field
(defop f-cc0 scmi)
(defop f-cc0 spsp)
(defop f-cc0 srsp)
(defop f-cc0 sbasp)
(endops)
;
;     Sequencer - four fields
;     field one- how to determine next instruction arithmetic
(defop f-sc0 seqsame)
(defop f-sc0 seqnext)
(defop f-sc0 seqskip)
(defop f-sc0 seqskip2)
(endops)
;
;     field two - next instruction arithmetic for alternate branch path
;     (same field values as f-sc0)
;
;
;     field three - kanban condition mask (so far, just a 0/1 test is
;     implemented in the simulator - and that's maybe all we need)
;
;     (princ "Defining sequencer masks ") (terpri)
;     (setq seqmaskzero 1) ; kanban boole value was zero
;     (setq seqmaskless 2) ; kanban boole value was <0 (high order bit set)
;     (setq seqmasknil 4) ; kanban boole value matched bitpattern for nil
;     (setq seqmasknetbusy 8); network can't accept another funcn - do it yourself
;     (setq seqmasknetavail 16); network has a job for you to accept
;

```

```

;
;   field four - conditional waits- including the conditional
;   wait being nonzero causes the sequencer to IGNORE an unready signal
;   from the device in question. Devices are DEFINED to not change
;   their outputs if commanded to IDLE.
(setq seqcwml 1)
(setq seqcwalu 2)
(setq seqcwbind 4)
;
;
;
; ....and NOW! A kludge to make opcode reversability and other
;   things possible. but, like, hey, we'll know better next time
;                                     -JCT
(defun dis-believe ()
  (endops)
  (defop f-m1c midle)      ; do nothing.
  (defop f-m1c mread)      ; read out of memory
  (defop f-m1c mwrite)     ; write
  (defop f-m1c mfadd)      ; fetch and add
  (endops);
  (defop f-m2c midle)      ; do nothing
  (defop f-m2c mread)      ; read out of memory
  (defop f-m2c mwrite)     ; write
  (defop f-m2c mfadd)      ; fetch and add
  (endops);
  (defop f-pc0 suidle)     ; entire stack unit is to idle.
  (defop f-pc0 snormal)    ;treat SP normally
  (defop f-pc0 subhead)    ;Use pvl instead of stackpointer
  (endops)
  (defop f-pc1 spop7)
  (defop f-pc1 spop6)
  (defop f-pc1 spop5)
  (defop f-pc1 spop4)
  (defop f-pc1 spop3)
  (defop f-pc1 spop2)
  (defop f-pc1 spop1)
  (defop f-pc1 sidle)
  (defop f-pc1 spush1)
  (defop f-pc1 spush2)
  (defop f-pc1 spush3)
  (defop f-pc1 spush4)
  (defop f-pc1 spush5)
  (defop f-pc1 spush6)
  (defop f-pc1 spush7)
  (defop f-pc1 spush8)
  (endops)
  (defop f-pc2 swidle)     ; note pwidle - Write idle !
  (defop f-pc2 swtop)      ; overwrite top element with PV0
  (defop f-pc2 swt-1)      ; overwrite t-1 with PV1
  (defop f-pc2 swboth)     ; overwrite both
  (endops)
  (defop f-pc3 spidle)     ; Don't write it back
  (defop f-pc3 spwrite)    ; Write final stptr value (after arithmetic) into PSP
  (endops)
  (defop f-rc0 suidle)     ; entire stack unit is to idle.
  (defop f-rc0 snormal)    ;treat SP normally
  (defop f-rc0 subhead)    ;Use rvl instead of stackpointer
  (endops)
  (defop f-rc1 spop7)
  (defop f-rc1 spop6)
  (defop f-rc1 spop5)
  (defop f-rc1 spop4)
  (defop f-rc1 spop3)
  (defop f-rc1 spop2)

```

```

(defop f-rc1 spop1)
(defop f-rc1 sidle)
(defop f-rc1 spush1)
(defop f-rc1 spush2)
(defop f-rc1 spush3)
(defop f-rc1 spush4)
(defop f-rc1 spush5)
(defop f-rc1 spush6)
(defop f-rc1 spush7)
(defop f-rc1 spush8)
(endops)
(defop f-rc2 swidle) ; note vWidle - Write idle !
(defop f-rc2 swtop) ; overwrite top element with RV0
(defop f-rc2 swt-1) ; overwrite t-1 with RV1
(defop f-rc2 swboth) ; overwrite both
(endops)
(defop f-rc3 spidle) ; Don't write it back
(defop f-rc3 spwrite) ; Write final stptr value (after arithmetic) into RSP
(endops)
(defop f-bc0 suidle) ; entire stack unit is to idle.
(defop f-bc0 snormal) ; treat SP normally
(defop f-bc0 subhead) ; Use bvl instead of stackpointer
(endops)
(defop f-bc1 spop7)
(defop f-bc1 spop6)
(defop f-bc1 spop5)
(defop f-bc1 spop4)
(defop f-bc1 spop3)
(defop f-bc1 spop2)
(defop f-bc1 spop1)
(defop f-bc1 sidle)
(defop f-bc1 spush1)
(defop f-bc1 spush2)
(defop f-bc1 spush3)
(defop f-bc1 spush4)
(defop f-bc1 spush5)
(defop f-bc1 spush6)
(defop f-bc1 spush7)
(defop f-bc1 spush8)
(endops)
(defop f-bc2 swidle) ; note bWidle - Write idle !
(defop f-bc2 swtop) ; overwrite top element with BV0
(defop f-bc2 swt-1) ; overwrite t-1 with BV1
(defop f-bc2 swboth) ; overwrite both
(endops)
(defop f-bc3 spidle) ; Don't write it back
(defop f-bc3 spwrite) ; Write final stptr value (after arithmetic) into BSP
(endops)
(defop f-scl seqsame)
(defop f-scl seqnext)
(defop f-scl seqskip)
(defop f-scl seqskip2)
(endops)
(dis-believe)

;
;
; Definitions for functional boxes to be used
; in the simulator
;
; Effaddr - how many bits are actually "seen" by a
; memory unit...
(defun addr-part (addr)

```



```

(extract addr '(32 0)))
;
;           Defining a memory box in terms of functionality
;
(defmacro memory-box (name output addr data control duse-mod)
  `(prog (effaddr)
    (pbbbp ',name)
    (setq effaddr (addr-part ,addr))
    (setq ,duse-mod (1+ ,duse-mod))
    (cond
      ((equal ,control midle)
        (setq ,duse-mod (1- ,duse-mod))
        (pbbb " idles"))
      ((equal ,control mread)
        (setq ,output (aref memory effaddr))
        (pbbb " reads"))
      ((equal ,control mwrite)
        (setf (aref memory effaddr) ,data)
        (setf ,output ,data)
        (pbbb " writes"))
      ((equal ,control mfadd)
        (setq ,output (aref memory effaddr))
        (setf (aref memory effaddr)
          (+ ,output
            ,data))
        (pbbb " fetch-and-adds"))
    )
  )
)
;
;           Defining a stack box in terms of functionality
;
(defmacro stack-box
  (name top utop spout tdata udata spsel motion write spreload duse-mod)
  `(prog ()
    ; first- select what to use as stackpointer
    (pbbbp ',name)
    (pbbbp " using")
    (setq ,duse-mod (1+ ,duse-mod))
    (cond
      ((equal ,spsel suidle) ; StackUnit Idle
        (pbbb " ... an idle cycle")
        (setq ,duse-mod (1- ,duse-mod))
        (go STACK-IDLE)) ; we're idling- so nothing changes.
      ((equal ,spsel snormal)
        (setq csp (addr-part ,spout))
        (pbbbp " old SP, "))
      ((equal ,spsel subhead)
        (setq csp (addr-part ,udata))
        (pbbbp " 2nd data as SP, "))
    )
    ; Second- effective SP motion control
    (pbbbp "SPdelta = ")
    (pbbbp (+ ,motion (- 0 sidle)))
    (setq csp (+ csp ,motion (- 0 sidle)))
    (pbbbp ", is now=")
    (pbbbp csp)
    (pbbbp ",")
    ;
    ; Third - write actionals
    (cond
      ((equal ,write swidle) (pbbbp " write none,"))
    )
  )
)

```

```

((equal ,write swtop)
 (setf (aref memory csp) ,tdata)
 (pbbbp " write top,"))
((equal ,write swt-1)
 (setf (aref memory (- csp 1)) ,udata)
 (pbbbp " write top-1,"))
((equal ,write swboth)
 (setf (aref memory csp) ,tdata)
 (setf (aref memory (- csp 1)) ,udata)
 (pbbbp "write both,"))
)
;      fourth- maybe reload the VSP?
(cond
 ((equal ,spreload spwrite)
  (setq ,spout csp)
  (pbbbp " new SP saved."))
 (t (pbbbp " keep old SP.")))
(pbbb " ")
;
;      last- output our data busses
(setq ,top (aref memory csp))
(setq ,utop (aref memory (- csp 1)))
STACK-IDLE
;      If idle, output DOES NOT change!
)
)

```



```

;      DISSIM.L    WSY@RPI-CS
;
; added - split binding into binding stack function and M2 memory function.
; added - fast branches via Shane/S-1 method of muxing next addr.
; added - kanban condition codes
; added - seqmasknil condition code using GC fields.
;
;      And now, what we all have been waiting for- The DIS Simulator:
;
;      <<< what an enormous cludge >>>
;
; (setq final-status t)
; (setq dis-graphics nil)
;
;      To make tracing easy, we define pbbb, which is
;      a function to print the blow-by-blow account of the run.
;      pbbb looks at the global blowbyblow to determine if
;      it should print or not
;
; (setq blowbyblow t)
; (defun pbbb (text)
;   (cond
;     (blowbyblow
;      (princ text)
;      (terpri)
;     )
;   )
; )
;
;      pbbbp prints text, no terpri, and returns the value of text.
; (defun pbbbp (text)
;   (cond
;     (blowbyblow
;      (princ text)
;      text
;     )
;     (t text)
;   )
; )
;
;      cycles is how many cycles to simulate.
; (defun disrun (cycles)
;   (stat-zero)
;   (catch 'branch-to-zero-halt
;     (prog (count)
;       (setq count cycles)
;       CYCLE
;       (setq count (- count 1))
;       (setq counter (- cycles count))
;       (one-cycle)
;       (cond ((> count 0) (go CYCLE)))
;     )
;   )(complete-status))
;
;      this is where the stats are zeroed out for each simulation
; (defun stat-zero ()
;   (setq d-time 0)
;   (setq dt-m0 0) (setq duse-m0 0.0)
;   (setq dt-m1 0) (setq duse-m1 0.0)
;   (setq dt-m2 0) (setq duse-m2 0.0)
;   (setq dt-vstk 0) (setq duse-vstk 0.0)
;   (setq dt-pstk 0) (setq duse-pstk 0.0)
;   (setq dt-rstk 0) (setq duse-rstk 0.0)
;   (setq dt-bstk 0) (setq duse-bstk 0.0)
;   (setq dt-seq 0) (setq duse-seq 0.0)
;   (setq dt-alu 0) (setq duse-alu 0.0)
;   (setq dt-net 0) (setq duse-net 0.0))

```

```

;
(defun one-cycle()
  (latch-the-CMI) ; CMI of current inst. is seen NOW, not next.
  (latch-bus-values)
  (cycle-m0)
  (cycle-m1)
  (cycle-m2)
  (cycle-vstk)
  (cycle-pstk)
  (cycle-rstk)
  (cycle-bstk)
  (cycle-alu)
  (cycle-net) ; NET MUST follow ALU- (because of status bits).
  (cycle-seq) ; ALU before sequencer or net- for status
               ; codes are "kanban"- available during...
               ; timing cludge- count a cycle only if
               ; we weren't in the loop-to-zero...
  (cond
    ((equal md0 0)
     (print-2-cr "Branching to zero - halting run." " ")
     (throw 'branch-to-zero-halt nil))
    (t (setq cycle-counter (+ cycle-counter 1))))
  (display-status)
)
;
;
; Busmux is the procedural equivalent of
; one column worth of crossbar switch. When executed, it
; selects one of the 16 distribution busses.
;
(defun busmux (bcode)
  (cond
    ((equal bcode b-md0) md0) ; M0 output
    ((equal bcode b-md1) md1) ; M1 output
    ((equal bcode b-md2) md2) ; M2 output
    ((equal bcode b-vd0) vd0) ; Top of VS
    ((equal bcode b-vd1) vd1) ; Undertop of VS
    ((equal bcode b-vsp) vsp) ; VS pointer
    ((equal bcode b-pd0) pd0) ; Top of PS
    ((equal bcode b-pd1) pd1) ; Undertop of PS
    ((equal bcode b-rd0) rd0) ; Top of RS
    ((equal bcode b-rd1) rd1) ; Undertop of RS
    ((equal bcode b-bd0) bd0) ; Top of BS
    ((equal bcode b-bd1) bd1) ; Undertop of BS
    ((equal bcode b-ad0) ad0) ; ALU output
    ((equal bcode b-sd0) sd0) ; SEQ next addr
    ((equal bcode b-nd0) nd0) ; NET output
    ((equal bcode b-cmi) cmi) ; CMI multiplexor
  )
)
;
; Kludge to allow negative CMI's, and mux the positive
; stackpointer onto the CMI dist'n bus. This is a hard
; bug to deal with in software bignums, so be careful.
;
(defun latch-the-CMI ()
  (setq cmi-control (s-cc0))
  (setq cv0 (s-cv0))
  (setq cmi
    (cond
      ((equal cmi-control scmi)
       (+
        (cond
          ((zerop (extract driven-instruction
            (list '1 (+ (car f-cv0) (cadr f-cv0)))))
            cv0)
          (t

```

```

        (- 0 cv0)))
        (cv1 (s-cv1))      ; pointer-count field
        (cv2 (s-cv2))      ; element-count field
    )
    ((equal cmi-control spsp) psp)
    ((equal cmi-control sbasp) bsp)
    ((equal cmi-control srsp) rsp)))
;
;       We latch bus values by assigning data values to module inputs
;       We have to latch them because not all
;       operations occur at the same time in this simulator.
;       Of course, this is a problem ONLY here in the simulator. Real
;       hardware would have no such problem.
(defmacro latch-an-input (inputname)
  '(let
    ((busmuxvalue
      (busmux
        (extract      ;we can generate the fieldname by looking at the input
          driven-instruction
          ,(implode    ; because the deffield macro creates
                    ; the field as f-<foo>
            (append
              (explode 'f-)
              (explode ',inputname)))))))
      (pbbbp ',inputname)(pbbbp " gets ")(pbbb busmuxvalue)
      (set ',inputname busmuxvalue)
    ))
  (defun latch-bus-values ()
    (progn ()
      (latch-an-input m0v0)
      (latch-an-input m0v1)
      (latch-an-input m1v0)
      (latch-an-input m1v1)
      (latch-an-input m2v0)
      (latch-an-input m2v1)
      (latch-an-input vv0)
      (latch-an-input vv1)
      (latch-an-input pv0)
      (latch-an-input pv1)
      (latch-an-input rv0)
      (latch-an-input rv1)
      (latch-an-input bv0)
      (latch-an-input bv1)
      (latch-an-input av0)
      (latch-an-input av1)
      (latch-an-input nv0)
      (latch-an-input sv0)
      ;
      ;       here- a cludge. The instruction
      ;       which is being driven just now already (supposedly)
      ;       has been mux-conditioned on sv1/sv2,
      ;       so only one n-bit wide mux is needed
      ;       This puts five gate delays in the sequencer,
      ;       but it speeds up branching tremendously.
      ;
      ;
      ;       ONLY HERE should SV2 ever be assigned to. In reality
      ;       SV2 is muxed with SV1 in the crossbar, on the basis of the
      ;       kanban condition codes. Heck, the sequencer has to wait for
      ;       M0 to respond anyway, so it may as well use the crossbar for
      ;       something useful.
      ;
      (latch-an-input sv1)
      (latch-an-input sv2)
    )
  )

```

```

)
;
;Definitions of the blocks start here!
;
;          m0
(defun cycle-m0 ()
  (setq m0op (s-m0c))
  (setq m0offset (s-m0v2))
  (setq m0effaddr (+ m0v0 m0offset))
  (memory-box
    M0
    MD0
    m0effaddr
    m0v1
    m0op
    duse-m0
  )
)
;
;          m1
(defun cycle-m1 ()
  (setq m1op (s-m1c))
  (setq m1offset (s-m1v2))
  (setq m1effaddr (+ m1v0 m1offset))
  (memory-box
    M1
    MD1
    m1effaddr
    m1v1
    m1op
    duse-m1
  )
)
;
;          m2
(defun cycle-m2 ()
  (setq m2op (s-m2c))
  (setq m2offset (s-m2v2))
  (setq m2effaddr (+ m2v0 m2offset))
  (memory-box
    M2
    MD2
    m2effaddr
    m2v1
    m2op
    duse-m2
  )
)
;
;          value stack
(defun cycle-vstk ()
  (setq spselect (s-vc0))
  (setq motion (s-vc1))
  (setq write (s-vc2))
  (setq spreload (s-vc3))
  (stack-box
    VS
    VD0
    VD1
    VSP
    VV0
    VV1
    spselect
    motion
    write
  )
)

```

```

    spreload
    duse-vstk
  )
;
;
;      Pending call stack (looks like pending, dont it?)
(defun cycle-pstk ()
  (setq spselect (s-pc0))
  (setq motion (s-pc1))
  (setq write (s-pc2))
  (setq spreload (s-pc3))
  (stack-box
    PS
    PD0
    PD1
    PSP
    PV0
    PV1
    spselect
    motion
    write
    spreload
    duse-pstk
  )
)
;
;
;      Return call stack
(defun cycle-rstk ()
  (setq spselect (s-rc0))
  (setq motion (s-rc1))
  (setq write (s-rc2))
  (setq spreload (s-rc3))
  (stack-box
    RS
    RD0
    RD1
    RSP
    RV0
    RV1
    spselect
    motion
    write
    spreload
    duse-rstk
  )
)
;
;
;      Binding stack (looks like the above, too. Maybe I should...)
(defun cycle-bstk ()
  (setq spselect (s-bc0))
  (setq motion (s-bc1))
  (setq write (s-bc2))
  (setq spreload (s-bc3))
  (stack-box
    BS
    BD0
    BD1
    BSP
    BV0
    BV1
    spselect
    motion
    write
  )
)

```



```

    spreload
    duse-bstb
  )
;
;
;           The ALU!
(defun cycle-alu()
  (setq alu-boole (s-ac0))
  (setq alu-select (s-ac1))
  (setq alu-format (s-ac2))
  (setq alu-comb (s-ac3))
  (setq alu-shift (s-ac4))
  ;
  ;       DANGER DANGER DANGER
  ;       We clear the sequencer status bits HERE!
  ;       Therefore, the network setting of a status bit
  ;       MUST happen after the ALU does it's work!
  (setq seqstatus 0)
  ;
  ;       first, do the "boole" calculation
  (setq alboole (boole alu-boole av0 av1))
  ;
  ;       Now we set the kanban condition codes on this result.
  ;       Since a boole-box has at most three gate delays, this
  ;       is OK to do (in terms of the hardware).
  (setq kanban
    (+
      (cond
        ((equal alboole 0) seqmaskzero)
        (t 0))
      (cond
        ((equal
          (extract alboole '(1 127))
          1) seqmaskless)
        (t 0))
      (cond
        ((and
          (equal
            (extract alboole f-cv1) 1); CV1 is pointer-count(ptr-length)
            (equal
              (extract alboole f-cv2) 0)); CV2 is total-count (struct-length)
          seqmasknil)
        (t 0))))
    (+
      (cond ((equal alboole 0) 1)
            (t 0))
      (cond ((extract alboole '(64 1)) 2)
            (t 0))))
  ;
  ;       now do the data selector
  (setq alsel
    (cond
      ((equal alu-select selzero) 0)
      ((equal alu-select sel1) 1)
      ((equal alu-select sel-1) -1)
      ((equal alu-select selav0) av0)))
  ;
  ;       now the combinational hardware ! LISP does the actual
  ;       bit twiddling for us. We can be mellow.
  (setq alcomb
    (cond
      ((equal alu-comb combadd)
        (+ alboole alsel))

```

```

((equal alu-comb combsubtract)
 (~ alboole alsel))
((equal alu-comb combmultiply)
 (* alboole alsel))
((equal alu-comb combddivide)
 (truncate (/ alboole alsel))))))

;
;
;   now we do the barrel shifter- for now, this is inefficient, but
;   it is what we need.
;   Maybe someday this will be fixed.
;
(setq ad0
 (ash alcomb alu-shift))

;
;   (+
;     (* (power 2 alu-shift) alcomb)
;     (quotient (* (power 2 alu-shift) alcomb) (power 2 256))))
;
;
;   network interface (not done yet)
(defun cycle-net() t)
;
;   Sequencer !
(defun cycle-seq()
 (setq seq-skip (s-sc0))
 (setq seq-altskip (s-sc1))
 (setq seq-condcode (s-sc2))
 (setq seq-waitmask (s-sc3))
 ;
 ;
 ;
 ;   Now we do the next instruction calculation
 ;   This part will get changed, no doubt. Among other
 ;   things, we may want to skip 0,1,2, or 3.
 ;
 (cond
 ((not (zerop (logand kanban seq-condcode)))
  (setq sd0 (+ sv2 seq-altskip))
  (pbbb "Sequencer using secondary address"))
 (t
  (setq sd0 (+ sv1 seq-skip))
  (pbbb "Sequencer using primary address"))
 )
 ;
 ;   Now, we send out the new instruction, and we're done.
 ;
 (setq driven-instruction sv0)
 )
;
(defun display-status ()
 (if dis-graphics (gr-update))
 (cond
 ((equal final-status 'short)
  (princ "Nextaddr (sd0) ") (print sd0) (terpri))
 (final-status
  (princ "Bus values at end of this cycle ") (terpri)
  (princ "M0 out   (md0) ") (princ md0) (terpri)
  (princ "M1 out   (md1) ") (princ md1) (terpri)
  (princ "M2 out   (md2) ") (princ md2) (terpri)
  (princ "V-top    (vd0) ") (princ vd0) (terpri)
  (princ "V-top-1  (vd1) ") (princ vd1) (terpri)
  (princ "V-stkptr (vsp) ") (princ vsp) (terpri)
  (princ "P-addr   (pd0) ") (princ pd0) (terpri)
 )
 )
 )

```

```

(princ "P-argptr (pd1) ") (princ pd1) (terpri)
(princ "R-addr (rd0) ") (princ rd0) (terpri)
(princ "R-argptr (rd1) ") (princ rd1) (terpri)
(princ "B-addr (bd0) ") (princ bd0) (terpri)
(princ "B-addr (bd1) ") (princ bd1) (terpri)
(princ "B-data (bd1) ") (princ bd1) (terpri)
(princ "ALU out (ad0) ") (princ ad0) (terpri)
(princ "Net data (nd0) ") (princ nd0) (terpri)
(princ "ImmData (cmi) ") (princ cmi) (terpri)
(princ "Nextaddr (sd0) ") (princ sd0) (terpri)
(princ "BSP: ") (princ bsp)
(princ " PSP: ") (princ psp)
(princ " RSP: ") (princ rsp) (terpri)
)
)
(defun complete-status ())
;
;
; Here's where we cold-boot the DIS machine-
; ALL initialization code goes here!
;
(defun cold-boot (boot-location &rest callargs)
;
; initialize stack pointers - these are busses too, but thats ok
(setq vsp (car v0))
(setq psp (car p0))
(setq rsp (car r0))
(setq bsp (car b0))
;
; Ancillaries- like driven-instruction
(setq driven-instruction (aref memory boot-location))
(setq md0 (aref memory (+ boot-location 1)))
(setq sd0 (+ boot-location 2))
(setq seqstatus 0)
(setq modulestatus 0)
(latch-the-cmi) ; Very important that CMI is prepped.
;
; (setq cycle-counter 0) ; how many cycles have we run?
;
; a "catcher" - where we loop on a branch to zero.
(setf (aref memory 0) 0) ; an infinitely looping location
(setf (aref memory 1) 0) ; an infinitely looping location
(setf (aref memory 2) 0) ; an infinitely looping location
;
; Build a call frame and arguments for our function to run...
(princ "Building fake call frame with argument list: ")
(princ callargs)
(prog ()
(setf (aref memory rsp) vsp) ; fake value stack frame start...
(setq rsp (+ 1 rsp))
(setf (aref memory rsp) 0) ; fake return location - infiloop
(setf (aref memory vsp) 'one-below)
MORE-ARGS
(cond
((null callargs)
(go NO-MORE-ARGS)))
(setq arg (car callargs))
(setq callargs (cdr callargs))
(setq vsp (+ vsp 1))
(setf (aref memory vsp) arg)
(go MORE-ARGS)
NO-MORE-ARGS
(setf (aref memory (+ vsp 1)) 'one-above)

```

```

)
;      initialize data busses - they all should be "right" now.
;
(setq md1 0)
(setq md2 0)
(setq vd0 (aref memory vsp))
(setq vd1 (aref memory (- vsp 1)))
(setq pd0 (aref memory psp))
(setq pd1 (aref memory (- psp 1)))
(setq rd0 (aref memory rsp))
(setq rd1 (aref memory (- rsp 1)))
(setq bd0 (aref memory bsp))
(setq bd1 (aref memory (- bsp 1)))
(setq ad0 0)
(setq nd0 0)
;
)

```



```
;          Functions to make Common Lisp look like Franz Lisp
(defun plus (x y) (+ x y))
;
(defun implode (str)
  (intern (coerce str 'string)))
;
(defun explode (str)
  (coerce (string str) 'list))
;
(defun putd (name func)
  (setf (symbol-function name) func))
;
(defun getd (name)
  (symbol-function name))
```



```

;          mcasmfuncs.lsp - WSY 3-Feb-86
;
;          Functions needed to successfully assemble output of
;          MC and MCA. (most of these functions just return 0).
;
;          A comment does nothing to the assembled instruction
;
; (defmacro comment (&rest foo)
; 0)
;
;          Locations *must be* handled before final assembly...
; (defmacro location (name &rest foo)
; 0)
;
;          Address-of is a transparent function for now. This
;          may change someday, if the linker gets more complicated.
; (defun address-of (name &rest foo)
;  name)
;
;          Crushmax's are meaningless at this point...
; (defmacro crushmax (count &rest foo)
; 0)
;
;          Reservations are useless now....
; (defmacro reserve (&rest foo)
; 0)
;
;          CMI - the CMI field is called CV0 in the simulator
; (defun cmi (x)
;  (cv0 x))

```



```

;
; DIS Micro Compiler - WSY - Nov 3, 1986
;
; 3-Nov-86: First version - build symbol table, generate calls to
; translink subr for all call references, generate
; call to lookup subr for symbol value references.
;
; 18-Nov-86: Second version - builds symbol table, still translink
; to calls, but creates symbol entries at compile time and
; hardcodes to symbol locations. Local (lexical) variables
; handled correctly for first 8 variables. DEFUN special form.
; Model: Instructions go in M0, symbol table in M1, everything
; else in M2. We only support conses and fixnums.
;
; 1-Dec-86: 2.5 version - Now Kanban detects NIL in hardware (using
; GC code information) so testing for NIL is much faster.
; (cond...) uses this new feature.
;
; 4-Dec-86: Symbol table finalized to contain the following-
; seq # --> +0 symbol-value-ptr
;          +1 symbol-function-ptr
;          +2 symbol-name-ptr
;          +3 symbol-plist-ptr
;          +4 symbol-package-ptr
;          +5-7 reserved for future expansion.
;
; 6-Jan-86: Added code for SETQ special form.
;
;
; initialize microcompiler to a completely clean slate.
(defun mc-hard-init ()
  (setq locals nil)
  (setq next-seq-num 992)
  (setq symbol-table
    '(t 1000 nil nil
      dis-nil 1008 nil nil
      *free-list* 1016 nil nil))
  (setq next-instr-addr 100)
  (app-res-initialize-machine-code)
)
;
;
; list of special forms - not all implemented
; just yet, but soon.
(setq *special-forms* '(
  defun ; working
  catch
  let ; working
  declare
  cond ; working
  go
  progn
  progv
  setq ; working
  throw
  unwind-protect
))
;
;
; list of "inline forms" - these are
; programs which are normal functions but
; the compiler implements inline for speed.
(setq *inline-forms* '(
  +
  *

```

```

-
/
1-
1+
car
cdr
eq
rplaca
rplacd
)

;
;
;           app-res - Append to Result
;           Bit of a kludge, used to append instructions
;           to the special variable "result"
;
(defmacro appres (inst)
  '(setq result
    (append result (list ,inst))))
;
;           The top of the compiler- the part you usually talk to
;           We compile the "implied prog" here- which is to
;           say, we set up the call, do the call, and if
;           that was not the last form in what we were handed,
;           we throw the result away (pop the stack) and
;           proceed with the next call)
;
(defun mc (form)
  (progv
    '(result)
    '(nil)
    (compile-form form)
    result)
  ; it's code, compile it.
  ; return the resulting assembly
)
;
;
;           Symbol table entries - just are a name, followed by
;           sequence-number symbol-value-ptr symbol-function-ptr.
;           (for now, at least) A null pointer means that the
;           pointer is unknown yet. (for the system builder to
;           use when loading)
;
;
(defun make-symbol-table-entry (entryname)
  (print-2-cr "Making a symbol table entry for " entryname)
  (cond
    ((member entryname symbol-table) t)
    (t
     (setq symbol-table
       (append symbol-table
         (list entryname
           (get-next-sequence-number)
           nil
           nil)
         )))))
;
;
;           Sequence numbers in this implementation are
;           equivalent to the runtime address of the
;           lambda-order control block (length 8)
;
(defun get-next-sequence-number ()
  (setq next-seq-num (+ next-seq-num 8)))
;
;           compile-implied-progn - this is how bodies of many
;           functions are compiled. The net result is
;           if there is one thing, return it. If more
;           than one thing, return the last. (go) not permitted
;           forms is a LIST of one or more forms -
;

```

```

;
;               NOT a form.  If it's a form, this dies!
;
(defun compile-implied-progn (forms)
  (print-2-cr "Compiling implied progn " forms)
  (compile-form (car forms)) ; compile the first form
  (cond
    ((null (cdr forms))
     (princ "implied progn done")
     (terpri))
    (t
     (princ "Not the last form, discarding return value")
     (terpri)
     (app-res-pop-vs-one) ; not the last form- throw away the value
     (compile-implied-progn (cdr forms)))
  )
)
;
;               compile a single form of any type- special, nonspecial, etc.
;
(defun compile-form (form)
  (cond
    ((listp form)
     (cond
       ((member (car form)
                *special-forms* )
        ; deal with special forms
        (compile-a-special-form form))
       ((member (car form)
                *inline-forms* )
        ; deal with known forms (inline)
        (compile-an-inline-form form))
       (t
        ; deal with non-special forms
        (compile-a-non-special-form form)
        )
      )
    )
    (t
     ; deal with lonely symbols
     (compile-value-reference form) ; note- not likely there will be
     ; local symbols- but it doesn't
     ; matter if we use compile-value-ref
    )
  )
)
;
;
;               print-2-cr - print 2 things, then Terpri
;
(defun print-2-cr (a b)
  (princ a)
  (princ b)
  (terpri)
)
;
;
;               How to compile a non-special form....
;
(defun compile-a-non-special-form (form)
  (print-2-cr "Preparing to call " (car form))
  (app-res-prepare-to-call-symbol (car form))
  (compile-push-args-on-vs form)
  (app-res-execute-prepared-call)
)
;
;
;               A general way to push all the args to a function...
;
(defun compile-push-args-on-vs (form)
  (prog (remaining)

```

```

(setq remaining form)
LOOP-FOR-ALL-ARGUMENTS
(setq remaining (cdr remaining))          ; get rid of CAR
(cond
  ((equal remaining nil) ;are we done yet?
   (go ARGS-NOW-ON-STACK))
  (t
   (compile-push-an-arg (car remaining)))
)
(go LOOP-FOR-ALL-ARGUMENTS)
ARGS-NOW-ON-STACK
(print-2-cr "All args on stack, emitting call" " ")
)
;
;
;
;
;           Push just one general argument...
(defun compile-push-an-arg (arg)
  (cond
    ((listp arg)      ;Do we have a function within?
     (compile-form arg))
    (t                ; Neither- just a value.
     (compile-value-reference arg))
  )
)
;
;           Compile-value-reference - get a value from somewhere
(defun compile-value-reference (symbol)
  (print-2-cr "Getting the value of " symbol)
  (cond
    ((member symbol locals)      ; Is it a local (lexical) variable, on the
                                ; stack, perhaps (eventually check for more
                                ; than 8 lexicals- but not yet).
     (app-res-vpush-from-local-values symbol))
    ((numberp symbol)           ; Numbers evaluate to themselves...
     (app-res-vpush-from-cmi symbol))
    (t                          ; Not a lexical variable, the value is
                                ; pointed to by the symbol table.
     (app-res-symbol-value-lookup
      symbol)                   ;value is put on VS
    )
  )
)
;
;
;           How to print out some code -
(defun printout (code)
  (setq ofile (open 'machine-code :direction :output
                   :if-exists :supersede))
  (setq *print-right-margin* 55)
  (pprint code ofile)
  (close ofile)
)

```



```

;          MCCRUSH.lsp - WSY 8-JAN-1987
;
;          Crush MC output into something that can
;          be expected to be runnable.
;
;          8-JAN-87 - V0.0 - must combine (location xxx) with
;          following instrs, put in sequencer instructions
;          IF NOT PRESENT.
;
;          3-Feb-87 - V1.0 - do some elementary crushing as well
;
;          16-Mar-87 -V1.5 - Crusher is getting sophisticated now.
;
(defun mca (code)
  (prog
    (result)
    (setq result code)
    (print-2-cr "Crushing instruction stream" " ")
    (setq result (crush-instruction-stream result))
    (print-2-cr "Inserting default opcodes" " ")
    (setq result (put-in-default-stuff result))
    (return result)
  )
)
;
;
;
;          Combine-labels is NOT used currently...we keep it around for
;          ol' time sake. Crush-instruction-stream does more and better.
;
(defun combine-labels (code)
  (prog
    (input result current-in current-out)
    (setq input code)
    (setq result nil)
    (setq current-out nil)
    (setq current-in nil)
    TOP-OF-LOOP
    (setq current-in (car input))
    (setq input (cdr input))
    (cond
      ((equal input nil)
       (go END-OF-LOOP))
      ((equal (caar current-in) 'location)
       (setq current-out (append current-out current-in)))
      (t
       (setq current-out (append current-out current-in))
       (setq result (append result (list current-out)))
       (setq current-out nil))
      )
    (go TOP-OF-LOOP)
    END-OF-LOOP
    (cond
      ((not (null current-in))
       (setq result (append result (list current-in))))
      (cond
        ((not (null current-out))
         (setq result (append result (list current-out))))
      )
    (return result)
  )
)
;
;
;

```



```

(setq dataflow-conflict-list ; Mustn't clobber a unit before it's used...
  (intersection busses1 units2))
(setq reservation-conflict-list ; What must be reserved...
  (intersection units1 reserves2))

;
;
; diagnostics-
(cond
  (nil
    (print-2-cr "Inst 1:" inst1)
    (print-2-cr "Units 1:" units1)
    (print-2-cr "Busses 1:" busses1)
    (print-2-cr "Inst 2:" inst2)
    (print-2-cr "Units 2:" units2)
    (print-2-cr "Busses 2:" busses2)
    (print-2-cr "Reserves 2:" reserves2)
    (print-2-cr "Resource Conflict:" resource-conflict-list)
    (print-2-cr "Sequence Conflict:" sequence-conflict-list)
    (print-2-cr "Dataflow Conflict:" dataflow-conflict-list)
    (print-2-cr "Reservation Conflict:" reservation-conflict-list)
  )
)

;
;
; If we have "crushmax 0" constraints, we CANNOT merge,
; we must append. So be it.
(cond
  ((member '(crushmax 0) inst2 :test 'equal) ; not allowed to crush...
    (print-2-cr "Latency constraint- control flow- " " no slack.")
    (return t)))

;
;
; If sequence-conflict-list is non-null, we know we can't
; combine anything...data flow prohibits it. So we just return T.
; Maybe someday we can look at the actual failure and with some
; hair, short-circuit some stack pushes (pushing onto the
; value stack and then using the value immediately is the most
; common form of sequence conflict.)
(cond
  (sequence-conflict-list
    (print-2-cr "Sequence clash on " sequence-conflict-list)
    (return t)))

;
;
; If resource-conflict is non-null, we know we can't just
; combine with ease. We have to look and see if it's a resource
; that can be combined (like a pair of push-one's combining into
; a push-two)
(cond
  (resource-conflict-list
    (print-2-cr "Resource-clash on " resource-conflict-list)
    (print-2-cr "Attempting resource-peephole fixup..." " ")
    (return (print (resource-peephole-fixup inst1 inst2)))))

;
;
; If reservation-conflict is non-null, a following instruction
; has a timing requirement that prohibits moving THIS instruction
; up at all. Hence, we combine it now and return it.
(cond
  (reservation-conflict-list
    (print-2-cr "Reservation clash on " reservation-conflict-list)
    (return (append inst1 inst2))))

;
;
; If dataflow-conflict is non-null, we cannot (must not!)
; move inst2 past inst1. Hence, we combine it now and return it.
(cond
  (dataflow-conflict-list
    (print-2-cr "Dataflow clash on " dataflow-conflict-list)
    (return (append inst1 inst2))))

;
;
; If we have "crushmax 1" constraints, we can append.

```

```

(cond
  ((member '(crushmax 1) inst2 :test 'equal) ; this is as far...
    (print-2-cr "Latency constraint- control flow- " " 1 unit slack.")
    (return (append inst1 inst2))))
;
;   If we got to here, apparently the two instructions have
;   almost nothing to do with each other and we can just
;   let them slide past one another. This should be rare but
;   we may as well do it if we can.
;   (print-2-cr "No conflicts found." " Instructions free to move")
;   (return nil)
)
)
;
;
(setq always-combinable
  '(
    comment
    location
    reserve
  )
)
;
;
(defun always-combinable-p (elem)
  (member (car elem) always-combinable))
;
;
;
(defun remove-always-combinables (list)
  (remove-if
    'always-combinable-p
    list)
)
;
;   Intersect- find the intersection of two lists.
;
;
(defun intersect (list1 list2)
  (progv
    '(targ)
    (list list2)
    (print "DARN! Still have a caller to INTERSECT")
    (remove nil
      (mapcar
        '(lambda (item1) (car (member item1 targ)))
        list1)
      )
    )
)
;
;   Given a control field, what module does it affect?
;   DANGER DANGER DANGER - we only look at the ZEROth control field!
;
;
(defun control-fields-to-modules (inst)
  (remove nil
    (mapcar 'from-xc0-yield-module inst)))
;
;
(defun from-xc0-yield-module (subop)
  (setq field (car subop))
  (cond
    ((equal field 'm0c) 'm0)
    ((equal field 'm1c) 'm1)
    ((equal field 'm2c) 'm2)
    ((equal field 'pc0) 'ps)
    ((equal field 'vc0) 'vs)
    ((equal field 'rc0) 'rs)
    ((equal field 'bc0) 'bs)
    ((equal field 'sc0) 'seq)
  )
)

```

```

((equal field 'crushmax) 'seq); crushmaxes are like seq instructions
((equal field 'ac0) 'alu)
((equal field 'nc0) 'net)
((equal field 'cmi) 'cmi)
(t nil)
)
)
;
;           Find out where our instruction gets it's data from.
;           We need this for the data-flow analysis that happens during crushing.
;
(defun busses-read-to-modules (inst)
  (remove nil
    (mapcar 'from-busses-yield-module inst)))
;
(defun from-busses-yield-module (subop)
  (setq buscode (cadr subop))
  (cond
    ((equal buscode 'b-md0) 'm0)
    ((equal buscode 'b-md1) 'm1)
    ((equal buscode 'b-md2) 'm2)
    ((equal buscode 'b-vd0) 'vs)
    ((equal buscode 'b-vd1) 'vs)
    ((equal buscode 'b-vsp) 'vs)
    ((equal buscode 'b-pd0) 'ps)
    ((equal buscode 'b-pd1) 'ps)
    ((equal buscode 'b-rd0) 'rs)
    ((equal buscode 'b-rd1) 'rs)
    ((equal buscode 'b-bd0) 'bs)
    ((equal buscode 'b-bd1) 'bs)
    ((equal buscode 'b-ad0) 'alu)
    ((equal buscode 'b-sd0) 'seq)
    ((equal buscode 'b-nd0) 'net)
    ((equal buscode 'b-cmi) 'cmi)
    (t nil) ; op not recognized, forget it.
  )
)
;
;
;           For simplicity, we break the incoming stream into segments before
;           applying the other rules. This is like basic-block manipulation.
;
(defun crush-instruction-stream (code)
  (reduce
    'append
    (mapcar
      'crush-a-segment
      (make-code-segments code))))
;
;
;           Knowing we're in a basic block, we can crush like crazy,
;           not having to worry about branch-ins and branch-outs.
;
(defun crush-a-segment (segment)
  (reduce
    'crush-instruction
    segment
    :initial-value '(())))
;
;
;
(defun crush-instruction (segment inst)
  (print-2-cr "Segment-so-far" segment)
  (print-2-cr "Next-instruction" inst)
  (prog
    (iclp)

```

```

(setq iclp (instr-clash-p ; look at the last old instr and the new one.
  (car (last segment))
  inst))
(cond
  ((equal iclp T)
    (print-2-cr "Appending" "...")
    (return (append segment (list inst))))
  ((equal iclp 'nil)
    (print-2-cr "No conflict" "...")
    (cond
      ((< (length segment) 2)
        (print-2-cr "short segment," "merging")
        (return (merge-with-last-inst segment inst)))
      (t ; not short- but no conflict at all
        (print-2-cr "moving up one, trying again" " ")
        (cond
          ((equal t ; Is it worth trying again?
            (instr-clash-p (car (last (butlast segment))) inst))
            (print-2-cr "Irresolvable clash one up," "merging")
            (return (merge-with-last-inst segment inst)))
          (t
            (print-2-cr "Move-up looks good, doing that" " ")
            (return (append
              (crush-instruction (butlast segment)
                (crushmax-peano-hack inst))
              (last segment)))))))
    (t
      (print-2-cr "Best we can do is a merge " "here.")
      (return
        (append (butlast segment) (list iclp))
      ))))
)
;
; Merge an instruction onto the end of a segment.
(defun merge-with-last-inst (segment inst)
  (append
    (butlast segment)
    (list
      (append
        (car (last segment))
        inst))))
)
;
; How we keep track of how far we've crushed this...
(defun crushmax-peano-hack (inst)
  (substitute
    '(crushmax 2)
    '(crushmax 3)
    (substitute
      '(crushmax 1)
      '(crushmax 2)
      (substitute
        '(crushmax 0)
        '(crushmax 1)
        inst :test 'equal)
      :test 'equal)
    :test 'equal))
)
;
; Make-code-segments - break a stream of instructions
; down into a list of "segments". A segment starts
; wherever, and ends at the end of stream or at a
; LOCATION, (including anonymous locations)
(defun make-code-segments (code)
  (reduce
    'code-segment-build
    (cdr code)
    :initial-value (list (list (car code)))
  )
)

```

```

)
;
;
(defun code-segment-build (built-part next-inst)
  (cond
    ((segment-terminator-p next-inst)
     (append built-part (list (list next-inst))))
    (t
     (append
      (butlast built-part)
      (list
       (append
        (car (last built-part))
        (list next-inst))))))
  )
)
;
;
(defun segment-terminator-p (inst)
  (or
   (find
    'location
    inst
    :test 'equal
    :key 'car)
  )
)

```



```

        END-LOOP
    )
;
;
(defun compile-%- (form)
  (compile-form (cadr form))
  (compile-form (caddr form))
  (app-res-subtract-vs-pair)
)
;
;
(defun compile-%* (form)
  (compile-form (cadr form))
  (compile-form (caddr form))
  (app-res-multiply-vs-pair)
)
;
;
(defun compile-%/ (form)
  (compile-form (cadr form))
  (compile-form (caddr form))
  (app-res-divide-vs-pair)
)
;
;
(defun compile-%eq (form)
  (compile-form (cadr form))
  (compile-form (caddr form))
  (app-res-VS-eq)
)
;
;
(defun compile-%car (form)
  (compile-form (cadr form))
  (app-res-VS-car)
)
;
;
(defun compile-%cdr (form)
  (compile-form (cadr form))
  (app-res-VS-cdr)
)
;
;
(defun compile-%l+ (form)
  (compile-form (cadr form))
  (app-res-subtract-vs-one))
;
;
(defun compile-%l- (form)
  (compile-form (cadr form))
  (app-res-subtract-vs-one)
)
;
;
(defun compile-%dlet (form)
  (compile-form (cadr (caadr form)))
  (app-res-bind-special-value (car (caadr form))))
;
;
(defun compile-%rplaca (form)
  (compile-form (cadr form))
  (compile-form (caddr form))
  (app-res-VS-rplaca)
)

```

```
;
;
(defun compile-trplacd (form)
  (compile-form (cadr form))
  (compile-form (caddr form))
  (app-res-VS-rplacd)
)
```



```

;
; Linker for DIS MC output WSY 2-17-87
;
;
; Input is a precrushed, default-inserted stream of
; instructions.
;
; Output is written directly into the DIS simulator
; memory. A second structure, an association
; list of locations, is also created and maintained.
;
; mclink - given an instruction stream, assemble them,
; taking note of LOCATIONS, and put them into memory.
; We supply a start address, mclink returns the next
; vacant address.
;
(defun mclink (stream startaddr)
  (prog (current-address instrs inst name value)
    (setq current-address startaddr)
    (setq instrs stream)
    SET-LOCATIONS ; first, go through and find all LOCATIONS
    (setq inst (car instrs))
    (setq instrs (cdr instrs))
    (check-for-location-info inst current-address)
    (setq current-address (+ current-address 1))
    (cond
      (instrs (go SET-LOCATIONS)))
      ; Done with address resolution, now for
      ; actual instruction calculation and
      ; placement.
    (setq current-address startaddr)
    (setq instrs stream)
    PLACE-AN-INSTRUCTION
    (setq inst (car instrs))
    (setq instrs (cdr instrs))
    (setq value ; what the actual bignum instruction is
      (reduce
        '+
        (mapcar
          'eval
          instrs)))
    (setf (aref memory current-address) value); put the instr in memory.
    (setq current-address (+ 1 current-address))
    (cond
      (instrs (go PLACE-AN-INSTRUCTION)))
    (return current-address)
  )
)
;
;
; check-for-location-info - SETQ location-names to
; address values. If it's a name in the symbol table,
; poke the DISSIM symbol table to be right, too.
;
;
(defun check-for-location-info (inst current-address)
  (cond
    ((setq name
      (find
        'location
        inst
        :test 'equal
        :key 'car))
      (set (cadr name) current-address)
      (update-dissim-symbol-table-for-function (cadr name) current-address)
    ))
  )
)
;

```

```
;
;      poke the DISSIM symbol table (kept in M0) to be right.
(defun update-dissim-symbol-table-for-function (symbol address)
  (cond
    ((member symbol symbol-table)
     (setf (aref memory (+ (symbol-seq-lookup symbol) 1)) address))))
```


Peephole optimization of instruction pairs.

We do this only during the crush phase. If we did it before, then we would close many sequence-clash-free instruction possibilities.

This version of peephole is table-driven. A list of valid transforms is given, and each is applied in sequence. This isn't the fastest but it certainly is the most flexible way (any function on a finite set can be defined by enumeration). For speed's sake, we run only those optimizations that are necessary- for instance, resource-conflicts. We do recurse on instr-clash-p to make sure the instructions don't have some other "funny" that prohibits recombination.

Like instr-clash-p, we allow coded returns-

T ----> No optimization found- you lose
list ----> the combined (optimized) instruction.

A peephole-options list looks like:

```
(
  ((instr-fragment-A)
   (instr-fragment-B)
   (instr-fragment-new)
   (lambda (A B NEW) (peephole-specific-code)))
  ...
)
```

When we find an instruction pair that contains ALL of instr-fragment-A and instr-fragment-B, we remove the fragments, and pass all three parts to the peephole-specific code for other peephole-specific manipulation. If we can manipulate successfully (and test via instr-clash-p recursively) we return the new instr. Otherwise, we return T (just like instr-clash-p).

```
(defun resource-peephole-fixup (inst-a inst-b)
  (prog (instr-pair)
    (cond ((not enable-resource-peepholes) (return t)))
    (print-2-cr "Resource-peephole A:" inst-a)
    (print-2-cr "Resource-peephole B:" inst-b)
    (setq instr-pair
      (reduce
        'peephole-once
        *resource-peephole-options* ; Mostly have to do with stacks...
        :initial-value (list inst-a inst-b)))
    (return
      (cond
        ((equal instr-pair (list inst-a inst-b))
         (print-2-cr "Resource peephole optimizer..." "fails to save.")
         t)
        (t
         (print-2-cr "Resource peephole optimizer..." "succeeds.")
         (cond
           ((equal t (instr-clash-p (car instr-pair) (cadr instr-pair)))
            (print-2-cr "but failed to do enough... conflict." " ")
            t)
           (t
            (print-2-cr "Resource peephole merge complete." " " ))
          ))
        ))
    )
```

```

;      (print-2-cr "Good new instruction is: "
;      (append (car instr-pair) (cadr instr-pair)))
;      (append (car instr-pair) (cadr instr-pair))))
;    ))))
;
;      Possible returns of peephole-once:
;      (list) ---> new instructions which will work if they
;      *can* be combined. Thus, you still have
;      to run instr-clash-p on these new instructions.
;      Only an instruction set that passes instr-clash-p
;      is gauranteed to work.
;
;      To avoid infinite recursion, don't have any optimizations
;      that can be repeated indefinitely.
(defun peephole-once (inst-pair option)
  (prog
    (inst-A inst-B frag-A frag-B frag-new)
    (setq inst-A (car inst-pair))
    (setq inst-B (cadr inst-pair))
    (setq frag-A (car option))
    (setq frag-b (cadr option))
    (return (cond
      ((and
        (subsetp frag-A inst-A :test 'equal)
        (subsetp frag-B inst-B :test 'equal))
        (eval (append
          (caddr option) ; the peephole-specific code
          (list (list 'quote
            (reverse (set-exclusive-or inst-a frag-a :test 'equal))))
            (list (list 'quote
            (reverse (set-exclusive-or inst-b frag-b :test 'equal))))
            (list (list 'quote
            (caddr option))) ; the new-inst-frag
          ))
        )
      (t
        inst-pair) ; We failed.... oh well.
    ))
  ))
;
;
;      DANGER DANGER DANGER
;
;      NEVER, EVER PUT IN A PEEPHOLE OPTION CONCERNING
;      TWO FIELDS THAT CAN SLIDE FREELY, UNLESS YOUR
;      INTENT IS THAT THEY SHALL NOT SLIDE PAST EACH
;      OTHER! THE PEEPHOLE CODE *MIGHT* MODIFY BOTH
;      INSTRUCTIONS BEFORE ANY OTHER OPTIMIZATIONS ARE DONE!
;
;      NEVER, EVER PUT IN A PEEPHOLE OPTION THAT MIGHT
;      INFINITELY LOOP. IF YOU DO THIS, YOU LOSE.
;
;      Of course, if this is your intent, go ahead. It's
;      your nickel...
;
;      (setq enable-resource-peeppholes T)
;      (setq *resource-peepphole-options*
;      '(
;      (
;      ((pah 1) (poo)) ; testing, testing, 1...2...3... testing.
;      ((zoo 3)) ; first instr. has?
;      ((forge)) ; 2nd instr. has?
;      (lambda (A B C) ; user data.
;      (list ; item-specific code to run

```



```

        (append A C) ; new first instruction.
        (append B 'foo)) ; new second instruction.
)
(
  ((vc0 snormal) ; Optimize two VPUSH1's into a VPUSH2
   (vc1 spush1) ; a vpush-1
   (vc2 swtop)
   (vc3 spwrite))
  ((vc0 snormal) ; another vpush-1
   (vc1 spush1)
   (vc2 swtop)
   (vc3 spwrite))
  ((comment "Combining two vpush-1's into a vpush-2")
   (vc0 snormal) ; combine to a vpush-2
   (vc1 spush2)
   (vc2 swboth)
   (vc3 spwrite))
  (lambda (A B C)
    (list
      (append
        (subst 'vv1 'vv0 A :test 'equal)
        C)
      B)
    )
  )
(
  ((bd0 b-md1) ; Optimize 2 BPUSH1's into a BPUSH2
   (bc0 snormal) ; 1st instr- pushing an old global value.
   (bc1 spush1)
   (bc2 swtop)
   (bc3 spwrite))
  ((bd0 b-cmi) ; 2nd instr - pushing next global address
   (bc0 snormal)
   (bc1 spush1)
   (bc2 swtop)
   (bc3 spwrite))
  ((comment "Combining two pushes of BStack into one")
   (bd0 b-cmi)
   (bd1 b-md1)
   (bc0 snormal)
   (bc1 spush2)
   (bc2 swboth)
   (bc3 spwrite))
  (lambda (A B C)
    (list
      (append A C)
      B)
    )
  )
(
  ((vc0 snormal) ; Optimize VPOP1-VPUSH1 into VOVERWRITE
   (vc1 spop1) ; 1st- the VPOP1
   (vc2 swidle)
   (vc3 spwrite))
  ((vc0 snormal) ; 2nd- the VPUSH1
   (vc1 spush1)
   (vc2 swtop)
   (vc3 spwrite))
  ((comment "Combining VPOP1 and VPUSH1 into VOVERWRITE")
   (vc0 snormal)
   (vc1 spidle)
   (vc2 swtop)
   (vc3 spidle))
  (lambda (A B C)
    (list

```

```
      A
      (append B C))      ; We do it this way because we must keep
                          ; VDO selection info valid!
    )
  )
))
```

Y	Y	EEEE	RRRR	AAA	ZZZZZ	U	U	N	N	III	SSSS
	Y	E	R R	A A	Z	U	U	N	N	I	S
Y	Y	E	R R	A A	Z	U	U	NN	N	I	S
	Y	EEEE	RRRR	A A	Z	U	U	N N	N N	I	SSS
	Y	E	R R	AAAAA	Z	U	U	N	NN	I	S
	Y	E	R R	A A	Z	U	U	N	N	I	S
	Y	EEEE	R R	A A	ZZZZZ	UUUUU	N	N	III	SSSS	

M	M	CCCC	SSSS	PPPP	EEEE	CCCC	III	AAA	L	SSSS						
MM	MM	C	S	P	P	E	C	I	A	A	L	S	SSSS			
M	M	M	C	C	S	P	P	E	C	C	I	A	A	L	S	SSSS
M	M	M	M	C	SSS	PPPP	EEEE	C	C	I	A	A	A	L	SSS	SSSS
M	M	M	M	C		P	E	C	C	I	AAAAA	L	L	S	SSSS	SSSS
M	M	M	M	C		P	E	C	C	I	A	A	A	L	S	SSSS
M	M	M	M	C	CCCC	SSSS	P	E	EEEE	CCCC	III	A	A	LLLLL	SSSS	SSSS

```

      L      SSSS PPPP ;; 1 999
      L      S    P  P  ;; 11 9 9
      L      S    P  P  ;; 1 9 9
      L      SSS PPPP ;; 1 9999
      L      S    P    ;; 1 9
      L      S    P    ;; 1 9
..   L      S    P    ; 1 9
..   LLLLL SSSS P      ; 111 999

```

Job DISBASE (810) queued to SYSSPRINT on 4-MAY-1987 10:54 by user YERAZUNIS,
UIC [YERAZUNIS], under account at priority 100, started on printer
CTHULUSCSA0: on 4-MAY-1987 11:02 from queue SYSSPRINT.

[illegible]

```

; Special Forms - how to compile them in the MC compiler
;
;
; How to compile a special form
;
(defun compile-a-special-form (form)
  (cond
    ((equal (car form) (quote defun)) ; Handle DEFUNs
     (compile-a-defun form))
    ((equal (car form) (quote cond)) ; Handle CONDs
     (compile-a-cond form))
    ((equal (car form) (quote setq)) ; Handle SETQs
     (compile-a-setq form))
    ((equal (car form) (quote let)) ; Handle LETs
     (compile-a-let form))
    (t
     (print-2-cr (car form)
                  " is a special form but I don't know how to deal with it ")
     )
  )
)
;
;
; How to deal with a DEFUN
;
(defun compile-a-defun (form)
  ; Defuns are handled wierd
  ; because they don't directly emit code, rather
  ; they set up tables. Then, we recursively descend
  ; to emit the code correctly. Hence, we can load
  ; files which contain immediate operations mixed
  ; with DEFUNs and it might be OK.
  (print-2-cr "Compiling a DEFUN for " (cadr form))
  (progv
    '(locals funcname) ; locals contains names of all vars
    '(nil nil nil) ; scoped locally- we get them via stack refs
    ; rather than symbol table extracts. Mucho
    ; faster.
    (setq locals (caddr form)) ; set local names up.
    (setq funcname (cadr form)) ; set local function name up.
    ;
    (print-2-cr "Local variables in this defun - " locals)
    (make-symbol-table-entry (cadr form)) ; put funcname in symbol table
    (app-res-symbol-function-location (cadr form) form); Insert marker for
    ; assembler to find start
    ; of code area
    (compile-implied-progn (caddr form)); and compile the body as an implied
    ; progn
  )
  (app-res-return-to-caller) ; now, we return to the caller
  ; of this function.
)
;
;
; How to compile a COND
;
; COND's are gooey because you have to jump somewhere if you get a
; test clause that is non-nil, but you don't do it right away, rather, you
; have to wait for the consequent to be executed, then you jump to the end.
; The simple way to handle it is to add an extra rez-up-execute-return to
; the end of the last consequent evaluation. Then you fake the return
; stack so that when that extra execute-return is executed, control appears
; to return to the instructions following the COND close.
; Faking this return stack is easy- we give COND it's own function frame

```

```

; so that it just returns. Then COND has control of what remains on
; the value stack, and no delicacy is necessary to deal with the faking
; of the return stack addresses.
; Note- eventually we can speed up returns by doing the delicate stuff.
;
(defun compile-a-cond (form)
  (print-2-cr "Compiling a COND which looks like " form)
  (prog
    (cond-exit) ; where the cond finally exits to,
    (nil) ; location yet unknown.
    (import (setq cond-exit (gensym "COND-EXIT-"))))
    ; We have to insert a (location cond-exit)
    ; at the end of the cond. Since cond-exit
    ; is dynamically bound, nested conds will
    ; work out fine.
    ;
    ; We now create a fake return frame - which
    ; when returned-from, will put execution at
    ; the end of the COND, but not alter RSP-1
    ; being the argument pointer. Remember,
    ; the PSP/RSP point to the top of VSP BEFORE
    ; the argumentlist began to get built.
    (print-2-cr "Creating fake return frame for COND termination" " ")
    ; Fake return frames look like real frames
    ; except they only update the SEQ
    ; information, not the VSP information.
    (app-res-create-fake-return-frame cond-exit)
    (prog (clauses this-clause cond-next)
      (setq clauses (cdr form))
      COMPILE-A-CLAUSE
      (cond
        ((null clauses) ; What to do if no clauses left:
          (go NO-MORE-CLAUSES))
        ((equal (caar clauses) t) ; a t-clause- which always executes...
          (setq this-clause (car clauses))
          (print-2-cr "Compiling t-clause: " this-clause)
          (cond
            ((null (cdr this-clause))
              (app-res-vpush-from-cmi t-value)); it might return t
            (t ; or it might return something else
              (compile-implied-progn (cdr this-clause))))
          (app-res-return-via-fake-frame); and this puts the right thing on the
          ; VS, thanks to the fake return frame.
          (go NO-MORE-CLAUSES))
        (t ; We have clauses left to evaluate-
          (setq this-clause (car clauses))
          (print-2-cr "Compiling clause: " this-clause)
          (compile-form (car this-clause)); execute the first clause
          (import (setq cond-next (gensym "COND-NEXT-"))))
          (app-res-branch-if-null cond-next)
          (cond
            ((null (cdr this-clause)) ; If no consequents, just branch to the
              ; cond-exit location (which takes care
              ; of what to return...)
              (app-res-return-to-caller)); fake return frame, remember.
            (t ; We *do* have consequent clauses, which we
              ; can execute as an implied progn. Then we
              ; go to the cond-exit location and that
              ; takes care of putting the return value in
              ; the right place.
              (app-res-pop-vs-one) ; Throw away the non-null predicate result
              (compile-implied-progn (cdr this-clause))
              (app-res-return-via-fake-frame))
            )
          )
          ; We're done with that clause...what's next?
          (app-res-symbol-function-location cond-next "Start of the next clause here")

```

```

; in case the clause wasn't satisfied, go
; here and try next clause...
(app-res-pop-vs-one) ; Throw away the annoying NIL from the
; previous clause (that didn't succeed)
(setq clauses (cdr clauses))
(go COMPILE-A-CLAUSE)
NO-MORE-CLAUSES ; If we fell thru to here, no clause was
; true. So, we have to return a NIL.
(print-2-cr "End of clauses - pushing the backing NIL" "")
(app-res-vpush-from-cmi (symbol-seq-lookup 'dis-nil)); push a NIL so that if
; no clause was satisfied, we have a NIL to
; return to the caller.
(app-res-pop-rs-two) ; we no longer need the fake frame.
; Return-via-fake gets rid of it, so we only
; do this popping when we don't do a
; return-via-fake.
(app-res-symbol-function-location cond-exit "Where the cond exits"))
)
;
;
;
;
; Compile-a-setq - compile SETQ's into actual changes
; in the data memory. We have to watch out for local variables
; (stack variables) though. Bound variables are OK, because even
; a bound variable has a symbol table entry...
;
(defun compile-a-setq (form)
  (print-2-cr "Compiling a SETQ which looks like " form)
  (prog (target value stuff)
    (setq stuff (cdr form))
    SETQ-LOOP
    (setq target (car stuff)) ; symbolname of where we put value
    (setq value (cadr stuff)) ; text of how we get value
    (setq stuff (cddr stuff)) ; deal with multiple setqs per call
    ;
    ; First, we get the value onto the stack.
    ; We do this by calling the compiler
    ; function COMPILE-FORM
    (compile-form value) ; ...value is now on the stack, either as
    ; a pointer to something, or as an immediate
    ; value. Either case is OK.
    ;
    ; Now we have to decide - where does this
    ; value go? It might be local/lexical or it
    ; might be special.
    (cond
      ((member target locals) ; ...are we lexical?
        (app-res-set-local-variable target)) ; note that we leave the value on
        ; the stack - the syntax of SETQ says we
        ; return the last value computed.
        ; ...we aren't lexical - symbol table change.
      (t
        (app-res-set-symbol-value target)))
    ; Check - are we done with all the stuff?
    ; If not, throw away top VS value and go
    ; around again. Else, we are done.
    (cond
      ((not (null stuff))
        (app-res-pop-vs-one) ; get rid of the value we don't want.
        (go SETQ-LOOP)) ; and go around again.
      (t
        )
      )
  )
)
;
;

```

```

;
;   LET's - handle variables of dynamic extent...
;   This requires doing five things:
;   1) saving an old value,
;   2) calculating and setting the new value,
;   2a) if more new values, recurse,
;   3) running the internal form,
;   4) restoring the old value,
;   5) returning the internal form.
;
;   We don't supply "full" common lisp LETting- in particular,
;   there must be at least one letted variable, and it MUST be
;   letted to a value (nil, if necessary). There are ways around
;   this that complicate the compiler but not the compiled code.
;
(defun compile-a-let (form &aux target value)
  (setq target (caaddr form))
  (setq value (cadr (caaddr form)))
  (cond
   ((member target locals)
    (print "I don't know how to do that yet"))
   (t
    ; special variable being bound...
    (app-res-bs-push-global-symbol target))) ; save them onto value stack
    ; recurse if necessary...
  (cond
   ((caddr form)
    (compile-form
     (append '(let) (list (cdadr form)) (caddr form)))))
    ; put the new value in...
  (compile-form
   (append '(setq) (list target) (list value))))
    ; If we're the innermost recursion,
    ; we can now run the forms-to-run...
  (cond
   ((null (cdadr form))
    (compile-form (caddr form))))
    ; Now, we unwind our saved values...
  (cond
   ((member target locals)
    (print "I don't know how to do that yet"))
   (t
    ; special variable being unbound...
    (app-res-global-unwind-bs-and-pop-two))))
;

```



```

;      Mcstubs.lsp - stubs to test MC with - WSY 3-11-86
;
;      Various functions to make the MC microcompiler work...
;
;      a default-but-explicit (sc0 seqnext) field is used
;      to tell the crusher NOT to optimize this instruction
;      over another possible branch/call/return.
;
;      In general, xC0 fields are used to indicate the x module
;      itself is busy this cycle.
;
;
(defun symbol-seq-lookup (symbol)
  (print-2-cr "Looking up sequence number for " symbol)
  (cond
    ((member symbol symbol-table)
     (cadr (member symbol symbol-table)))
    (t
     (make-symbol-table-entry symbol)
     (symbol-seq-lookup symbol)))
  )
;
;
;
(defun app-res-initialize-machine-code ()
  (print-2-cr "Initialize-machine here" " .")
  )
;
;
(defun app-res-vpush-from-cmi (cmi)
  (print-2-cr "app-resing a push to VS from cmi with value..." cmi)
  (appres
   (append text-vpush-from-cmi ; VPUSH1 returns the nonchanging part..
    (list (append '(cmi) (list cmi)))))
  )
(setq text-vpush-from-cmi
  '(
    (comment "Pushing a CMI value onto VS")
    (vv0 b-cmi) ; stack gets input from CMI
    (vc0 snormal) ; use normal VSP
    (vc1 spush1) ; push 1
    (vc2 swtop) ; write top
    (vc3 spwrite))) ; save new stackpointer
  )
;
;
;
(defun app-res-prepare-to-call-vs-addr ()
  (princ "app-resing a prepare-to-call with VS having address.")
  (terpri)
  (appres text-prepare-to-call-vs-addr-1)
  (appres text-prepare-to-call-vs-addr-2)
  )
(setq text-prepare-to-call-vs-addr-1
  '(
    (comment "Preparing to call, addr is now on VS")
    ; Store the new address.....
    (pv0 b-vd0) ; stacktop is addr to call
    (pv1 b-vsp) ; undertop is VSP at call
    (pc0 snormal) ; use normal PSP
    (pc1 spush2) ; push 2
    (pc2 swboth) ; write both
    (pc3 spwrite) ; save new PSP
  )
)

```



```

    (comment "Popping VS once")
    (vc0 snormal)      ; use normal VSP
    (vc1 spop1)        ; pop one
    (vc2 swidle)        ; don't write anything
    (vc3 spwrite)       ; save VSP back
  ))
;
;   pop-rs-two - throw away the most recent return frame.
;   THIS IS BAD TO DO IF IT WASN'T A FAKE FRAME WE ARE THROWING AWAY!
;
(defun app-res-pop-rs-two ()
  (appres text-pop-rs-two)
)
(setq text-pop-rs-two
  '(
    (comment "Popping a fake return frame off of RS")
    (rc0 snormal)      ; use normal RSP
    (rc1 spop2)        ; pop two
    (rc2 swidle)        ; write nothing
    (rc3 spwrite)       ; save RSP back
  )
)
;
; To return to our caller, we just pop the return location off of the
; return stack.  Because our functions are defined to "eat" the data
; off of the call only for locally wired functions, and some functions
; don't remove the data, we (to insure that the right result gets returned)
; have to move the current VS top to the location pointed to in the
; return stack (thereby smashing the top of the argument list but who
; cares any more, we're returning and that local context is vapor now anyway.
;
(defun app-res-return-to-caller ()
  (appres text-return-to-caller-1)
  (appres text-return-to-caller-2)
  (appres-anonymous)
)
(setq text-return-to-caller-1
  '(
    (comment "Return to caller using RS data")
    (comment "No branch permitted - return latency 2")
    (crushmax 2)        ; We can move this up at most two instrs.
    (reserve seq vs rs) ; and we must reserve the seq, vs, and rs.
    (m0v0 b-rd0)        ; M0 start fetching new instr. stream,
    (m0c mread)         ; M0 fetches instr,
                        ; Sequencer gets a new next-address.
    (sv0 b-md0)         ; instructions come from M0 still,
    (sv1 b-rd0)         ; but active addr comes from return stack,
    (sv2 b-rd0)         ; (branches not permitted),
    (sc0 1)             ; add 1 for next address(normal),
    (sc1 1)             ; add 1 for next addr. (alternate),
    (sc2 0)             ; no conditions for branch tested,
    (sc3 0)             ; don't ignore any module-not-ready signals
  )
)
;
;
(setq text-return-to-caller-2
  '(
    (comment "No branch permitted - return latency 1")
    (crushmax 1)
    (sc0 seqnext)        ; a sequencer op- to prevent crushing
                        ; Put the func. result where it's expected.
    (vv0 b-vd0)         ; value stack- take your current top;
    (vv1 b-rd1)         ; and using return stack top,
    (vc0 subhead)        ; for the stackpointer,
    (vc1 spush1)         ; point it to the first local frame loc.
  )
)

```

```

(vc2 swtop)          ; write the top,
(vc3 spwrite)        ; and save the new VSP.
                     ; Get rid of the return frame on RS
(rc0 snormal)        ; use standard stackpointer,
(rc1 spop2)          ; pop 2 elements,
(rc2 swidle)         ; no writing now,
(rc3 spwrite)        ; but save the new RSP.
)
;
;
; Return-via-fake-frame is a lot like returning to a caller,
; don't bother fixing up the VSP. Just really branch to the
; location pointed to by the RS.
;
(defun app-res-return-via-fake-frame ()
  (appres text-return-via-fake-frame-1)
  (appres text-return-via-fake-frame-2)
  (appres-anonymous)
)
(setq text-return-via-fake-frame-1
  '(
    (comment "Return to faked caller using RS data")
    (comment "No branch permitted - return latency 2")
    (crushmax 2)          ; We can move this up at most two instrs.
    (reserve seq rs)      ; and we must reserve the seq and rs.
    (m0v0 b-rd0)          ; M0 start fetching new instr. stream,
    (m0c mread)           ; M0 fetches instr,
                        ; Sequencer gets a new next-address.
    (sv0 b-md0)           ; instructions come from M0 still,
    (sv1 b-rd0)           ; but active addr comes from return stack,
    (sv2 b-rd0)           ; (branches not permitted),
    (sc0 1)               ; add 1 for next address(normal),
    (sc1 1)               ; add 1 for next addr. (alternate),
    (sc2 0)               ; no conditions for branch tested,
    (sc3 0)               ; don't ignore any module-not-ready signals
  )
)
(setq text-return-via-fake-frame-2
  '(
    (comment "No branch permitted - return latency 1")
    (crushmax 1)
    (sc0 seqnext)         ; a sequencer op- to prevent crushing
                        ; Put the func. result where it's expected.
    (rc0 snormal)        ; use standard stackpointer,
    (rc1 spop2)          ; pop 2 elements,
    (rc2 swidle)         ; no writing now,
    (rc3 spwrite)        ; but save the new RSP.
  )
)
;
;
; vpush-from-local-values - gets a value that's local
; (i.e. was passed as a parameter) and pushes it onto the
; value stack. This works only for the first 7 arguments.
; The argument number is compiled into the stack motion.
;
(defun app-res-vpush-from-local-values (symbol)
  (print-2-cr "Getting a local variable from the value stack " symbol)
  (prog (displacement)
    (setq displacement (position symbol locals))
    (appres
      (append
        (list

```



```

        (vc0 snormal)      ; use standard stackpointer,
        (vc1 spush1)      ; add 1 to it,
        (vc2 swtop)       ; write a word on top of stack,
        (vc3 spwrite)     ; and keep the new stack pointer.
    )
)
;
; symbol-function-lookup - Given a symbol table entry,
; what is the function pointer now bound to that symbol?
; For this, we look in the runtime symbol table, where
; location+1 contains a pointer to the function. We
; leave the pointer to the function on top of VS
;
(defun app-res-symbol-function-lookup (symbol)
  (appres
    (append
      (list
        (list
          'comment
          "Getting entry-point of symbol"
          symbol
        )
        (list (list 'cmi (symbol-seq-lookup symbol)))
        text-symbol-function-lookup-1
      )
    )
    (appres
      text-symbol-function-lookup-2
    )
  )
  (setq text-symbol-function-lookup-1
    '(
      (comment "Cycle M1 from CMI addr +1 offset and get absolute addr")
      (mlv0 b-cmi)      ; address on CMI
      (mlv2 1)          ; offset +1
      (mlc mread)       ; tell me what's there
    )
  )
  (setq text-symbol-function-lookup-2
    '(
      (comment "Take the pointer M1 gave us, put it on VS")
      (vv0 b-md1)       ; take your data from M1,
      (vc0 snormal)     ; use standard stackpointer,
      (vc1 spush1)      ; add 1 to it,
      (vc2 swtop)       ; write a word on top of stack,
      (vc3 spwrite)     ; and keep the new stack pointer.
    )
  )
)
;
;
;
; Creating a fake return frame is fairly easy. We just
; do as we did for a real execute-call but we take the
; address we return to from the CMI rather than the seq.
;
(defun app-res-create-fake-return-frame (location)
  (princ "Creating a fake return frame")
  (appres
    (append
      (list '(comment "Creating a fake return frame"))
      (list (list 'cmi location))
      text-create-fake-return-frame
    )
  )
  (setq text-create-fake-return-frame
    '(
      (rv0 b-cmi)      ; fake return address is on CMI,
      (rv1 b-rd1)      ; We must keep the old argptr valid for locals
      (rc0 snormal)    ; use the normal RSP,
      (rc1 spush2)     ; push 2 values,
      (rc2 swboth)     ; write 'em both,
    )
  )
)

```



```

    (rc3 spwrite)          ; keep the new RSP.
))
;
;
;   app-res-bs-push-global-symbol - push the
;   value of and address of a global symbol
;
(defun app-res-bs-push-global-symbol (target &aux seqnum)
  (setq seqnum (symbol-seq-lookup target))
  (appres
    (append
      (list (list 'Comment "Saving old value of" target))
      (list (list 'cmi seqnum))
      text-bs-push-global-1))
    (appres
      text-bs-push-global-2))
  (setq text-bs-push-global-1
    '(
      (mlv0 b-cmi)          ; get address from CMI,
      (mlc mread)          ; read what's there - that's symbol's value.
      (bd0 b-cmi)          ; take the CMI address, push it on BS,
      (bc0 snormal)        ; use standard binding stack pointer,
      (bc1 spush1)         ; push one element,
      (bc2 swtop)          ; write the top,
      (bc3 spwrite)        ; keep new stackpointer.
    ))
  (setq text-bs-push-global-2
    '(
      (comment "Push the old global value onto binding stack")
      (bd0 b-md1)          ; here's what that global had as a value,
      (bc0 snormal)        ; use the standard BSP,
      (bc1 spush1)         ; push the value from the symbol table,
      (bc2 swtop)          ; write the one element,
      (bc3 spwrite)        ; save the new stackpointer.
    ))
;
;   app-res-unwind-bs-and-pop-two - uses an address/value pair
;   to undo a binding on the BS.
;   This code *will* work on things which really exist on the
;   value stack, M2, or somewhere else... but in keeping with
;   the model for this compiler, we don't make that kind of
;   back-door bus reference. Hence, there is a global-unwind,
;   a local-unwind, and maybe an array-unwind (only if we implement
;   full SETF compatibility for LETs.)
(defun app-res-global-unwind-bs-and-pop-two ()
  (appres text-global-unwind-bs-pop-two-1)
)
(setq text-global-unwind-bs-pop-two-1
  '(
    (comment "Unbinding a special variable")
    (mlv0 b-bd1)          ; the address to smash,
    (mlv1 b-bd0)          ; what to smash it to,
    (mlc mwrite)          ; write that symbol table.
    (bc0 snormal)        ; use normal BSP,
    (bc1 spop2)           ; pop down two,
    (bc2 swidle)          ; don't write anything,
    (bc3 spwrite)        ; save new stackpointer.
  ))
;
(defun app-res-branch-to-location (target-location commentary)
  (appres
    (append
      (list
        '(comment "Unconditional branch"))
      (list

```

```

        (list 'comment commentary))
      (list
        (list 'cmi target-location))
        text-branch-to-location-1))
    (appres
      text-branch-to-location-2)
    (app-res-anonymous)
  )
  (setq text-branch-to-location-1
    '(
      (comment "No branch permitted - branch latency 2")
      (crushmax 2)
      (reserve seq)           ; reserve the sequencer as well...
      (sc0 seqnext)          ; we are to add 1 to get next addr
                              ; M0 should also get addr from CMI,
      (m0v0 b-cmi)           ; address from CMI,
      (m0c mread)            ; read it.
    ))
  (setq text-branch-to-location-2
    '(
      (comment "No branch permitted - branch latency 1")
      (crushmax 1)
      (sc0 seqnext)
    )
  )
  ;
  ;
  ;
  (defun app-res-branch-if-null (target-location)
    (appres
      (append
        (list
          (list 'comment "Branch-if-null to " target-location))
          (list
            (list 'cmi target-location))
            text-branch-if-null-1))
        (appres
          text-branch-if-null-2)
        (appres
          text-branch-if-null-3)
        (app-res-anonymous)
      )
      (setq text-branch-if-null-1
        '(
          (comment "No branch permitted - cond branch latency 3")
          (crushmax 3)
          (reserve seq)
          (av0 b-vd0)          ; look at the top of value stack,
          (ac0 boola)          ; just pass thru the "a" value,
                              ; so we get the Kanban set on it's nilness
          (sv1 b-sd0)          ; this is the "normal" address,
          (sv2 b-cmi)          ; and this is the alternate,
          (sc0 seqnext)         ; next primary instruction addr n+1,
          (sc1 seqsame)         ; next alt instruction addr is this address,
          (sc2 seqmasknil)     ; take alternate if alubooole is nil.
        )
      )
      (setq text-branch-if-null-2
        '(
          (comment "No branch permitted - cond branch latency 2")
          (crushmax 2)
          (comment "if branching, M0 sees new address at start this cycle")
          (sc0 seqnext)         ; a sequencer op to prevent crushing
        )
      )
      (setq text-branch-if-null-3

```

```

' (
  (comment "No branch permitted - cond branch latency 1")
  (comment "if branching, M0 returned new instr. stream to seq")
  (crushmax 1)
  (sc0 seqnext)          ; a sequencer op to prevent crushing
)
;
;
;
;
;      Stubs for inline code go here... if they may be
;      universally applicable. Otherwise, they go in the inline
;      file.
;
;
;      Add two fixnums
(defun app-res-add-VS-pair ()
  (print-2-cr "Adding top two elements of VS" " ")
  (appres text-add-vs-pair-1)
  (appres text-pop-VS-1-overwrite-ALU)
)
;
(setq text-add-vs-pair-1
  '(
    (comment "Adding top two elements of VS in ALU")
    (av0 b-vd0)          ; one arg is top of VS
    (av1 b-vd1)          ; second arg is one from top of VS
    (ac0 boolb)          ; ...here's av1
    (ac1 selav0)          ; ...here's av0
    (ac2 fmtfix)          ; fixnum format
    (ac3 combadd)         ; add the numbers
    (ac4 shift0)         ; don't shift the result
  )
)
;
(setq text-pop-VS-1-overwrite-ALU
  '(
    (comment "Now put result back on value stack")
    (vv0 b-ad0)          ; read data from the ALU
    (vc0 snormal)         ; use normal VSP
    (vc1 spopl)           ; go down one below top of stack
    (vc2 swtop)           ; write over the top of stack
    (vc3 spwrite)         ; save the new VSP
  )
)
;
;      Subtract two fixnums
(defun app-res-subtract-VS-pair ()
  (print-2-cr "Subtracting top two elements of VS" " ")
  (appres text-subtract-vs-pair-1)
  (appres text-pop-VS-1-overwrite-ALU)
)
;
(setq text-subtract-vs-pair-1
  '(
    (comment "Subtract top two elements of VS in ALU")
    (av0 b-vd0)          ; one arg is top of VS
    (av1 b-vd1)          ; second arg is one from top of VS
    (ac0 boolb)          ; ...here's av1
    (ac1 selav0)          ; ...here's av0
    (ac2 fmtfix)          ; fixnum format
    (ac3 combsubtract)    ; subtract the numbers
    (ac4 shift0)         ; don't shift the result
  )
)
;

```

```

;           Multiply two fixnums
(defun app-res-multiply-VS-pair ()
  (print-2-cr "Multiplying top two elements of VS" " ")
  (appres text-multiply-vs-pair-1)
  (appres text-pop-VS-1-overwrite-alu)
)
;
; (setq text-multiply-vs-pair-1
; '(
;   (comment "Multiplying top two elements of VS in ALU")
;   (av0 b-vd0)           ; one arg is top of VS
;   (av1 b-vd1)           ; second arg is one from top of VS
;   (ac0 boolb)           ; ...here's av1
;   (ac1 selav0)          ; ...here's av0
;   (ac2 fmtfix)          ; fixnum format
;   (ac3 combmultiply)    ; multiply the numbers
;   (ac4 shift0)          ; don't shift the result
; )
;
;           Divide two fixnums
(defun app-res-divide-VS-pair ()
  (print-2-cr "Dividing top two elements of VS" " ")
  (appres text-divide-vs-pair-1)
  (appres text-pop-VS-1-overwrite-ALU)
)
;
; (setq text-divide-vs-pair-1
; '(
;   (comment "Dividing top two elements of VS in ALU")
;   (av0 b-vd0)           ; one arg is top of VS
;   (av1 b-vd1)           ; second arg is one from top of VS
;   (ac0 boolb)           ; ...here's av1
;   (ac1 selav0)          ; ...here's av0
;   (ac2 fmtfix)          ; fixnum format
;   (ac3 combddivide)     ; add the numbers
;   (ac4 shift0)          ; don't shift the result
; )
;
;           Add one to a fixnum
(defun app-res-add-VS-one ()
  (print-2-cr "Adding one to top of VS" " ")
  (appres text-add-vs-one-1)
  (appres text-vs-overwrite-ALU)
)
;
; (setq text-add-vs-one-1
; '(
;   (comment "Adding one to top of VS with ALU")
;   (av1 b-vd0)           ; one arg is top of VS
;                           ; AV0 is not connected to anything
;   (ac0 boolb)           ; ...here's AV1
;   (ac1 sell)            ; ...here's the +1
;   (ac2 fmtfix)          ; fixnum format
;   (ac3 combadd)         ; add the numbers
;   (ac4 shift0)          ; don't shift the result
; )
;
; (setq text-vs-overwrite-alu
; '(
;   (comment "Now put result back on value stack")
;   (vv0 b-ad0)           ; read data from the ALU
;   (vc0 snormal)         ; use normal VSP
;   (vc1 spidle)          ; don't move pointer, keep it
; )

```

```

        (vc2 swtcp)          ; write over the top of stack
        (vc3 spidle)        ; save the new VSP
    )
;
;      Subtract one from a fixnum
(defun app-res-subtract-vs-one ()
  (print-2-cr "Subtracting one from VStop" " ")
  (appres text-subtract-vs-one-1)
  (appres text-vs-overwrite-alu)
)
;
(setq text-subtract-vs-one-1
  '(
    (comment "Subtract one from VStop in ALU")
    (avl b-vd0)          ; one arg is top of VS- in with AV1
                        ; AV0 is not used here
    (ac0 boolb)          ; ...Here's avl
    (ac1 sel-1)          ; ...here's a -1
    (ac2 fmtfix)         ; fixnum format
    (ac3 combadd)        ; add the numbers (which subtracts one)
    (ac4 shift0)         ; don't shift the result
  )
)
;
;
;
;
(defun app-res-VS-car ()
  (print-2-cr "Taking CAR of top VS element inline" " ")
  (appres text-vs-car-1)
  (appres text-vs-overwrite-M2)
)
;
(setq text-vs-car-1
  '(
    (comment "Taking the car of whatever's on VS top")
    (m2v0 b-vd0)        ; get what's on VS top
    (m2c mread)         ; what do we have at the car
  )
)
;
(setq text-vs-overwrite-M2
  '(
    (comment "Shove M2 result back onto VS, by overwriting")
    (vv0 b-md2)          ; MD2 has car,
    (vc0 snormal)        ; use standard VSP,
    (vc1 sidle)          ; no pointer motion,
    (vc2 swtop)          ; overwrite top- new data replaces old,
    (vc3 spidle)         ; no need to write back- it's not changed.
  )
)
;
;
(defun app-res-VS-cdr ()
  (print-2-cr "Taking CDR of top VS element inline" " ")
  (appres text-vs-cdr-1)
  (appres text-vs-overwrite-M2)
)
;
(setq text-vs-cdr-1
  '(
    (comment "Taking the cdr of whatever's on VS top")
    (m2v0 b-vd0)        ; get what's on VS top
    (m2v2 1)            ; CDR's are next word
    (m2c mread)         ; what do we have at the cdr
  )
)
;

```

```

;
;
(defun app-res-vs-eq ()
  (print-2-cr "Doing an EQ on the VS" " ")
  (appres text-vs-eq-1)
  (appres text-vs-eq-2)
  (appres text-vs-eq-3)
  (appres text-vs-eq-4)
  (app-res-anonymous)
)
(setq text-vs-eq-1
  '(
    (comment "Checking VS and VS-1 for EQness")
    (comment "No branch permitted - skip latency 3")
    (reserve seq alu vs) ; we'll need these
    (crushmax 3) ; can't move up too far,
    ; Route VS and VS-1 to ALU
    (av0 b-vd0) ; VS top
    (av1 b-vd1) ; VS top -1
    (ac0 boolxor) ; XOR them
    ; may as well ignore the rest of the ALU...
    (sv1 b-sd0) ; normal next addr- no skip
    (sv2 b-sd0) ; same normal next addr- but we'll skip.
    (sc0 seqnext) ; if not EQ, no skip, T will be overwritten
    (sc1 seqskip) ; if EQ, skip an instruction, and no
    ; overwriting of the T with NIL
    (sc2 seqmaskzero) ; skip if zero enabled.
  )
)
(setq text-vs-eq-2
  '(
    (comment "We pop the VS, and overwrite a T onto it. ")
    (comment "No branch permitted - skip latency 2")
    (reserve seq alu vs) ; needed later.
    (crushmax 2) ; maximum movement
    (vv0 b-cmi) ;
    (vc0 snormal) ; use the normal stackpointer,
    (vc1 spopl) ; go down one entry,
    (vc2 swtop) ; overwrite it with vv0.
    (vc3 spwrite) ; save the new stackpointer
    (cmi 1000) ; Symbol table 1'st entry is T
    (cv1 1) ; a pointer, one of them,
    (cv2 1) ; total length, one.
  )
)
(setq text-vs-eq-3
  '(
    (comment "Here we sit and wait. Nothing to do.")
    (comment "No branch permitted - skip latency 1")
    (reserve seq alu vs) ; needed real soon now
    (crushmax 1) ; maximum movement
  )
)
(setq text-vs-eq-4
  '(
    (comment "This instruction may or may not be executed. Depends.")
    (comment "No branch permitted, skip latency 0 ??? ")
    (comment "It's here we maybe push a NIL")
    (reserve seq alu vs)
    (crushmax 1)
    (cmi 0) ; CMI for nil
    (cv1 1) ; one pointer
    (cv2 0) ; but no data. That's nil.
    (vv0 b-cmi) ; Proceed to push NIL,
    (vc0 snormal) ; use normal stackpointer,
    (vc1 sidle) ; don't move it,
  )
)

```

```

        (vc2 swtop)          ; overwrite top,
        (vc3 spidle)        ; don't bother to resave unchanged VSP.
    )
;
;
;
(defun app-res-vs-rplaca ()
  (print-2-cr "Doing a RPLACA on VS" " ")
  (appres text-vs-rplaca-1)
)
(setq text-vs-rplaca-1
  '(
    (comment "RPLACAing Vunder with Vtop")
    (comment "Since you can *only* rplaca conses, use M2")
    (m2v0 b-vd0)          ; stacktop is value to poke,
    (m2v1 b-vd1)          ; undertop is where to poke it,
    (m2v2 0)              ; offset zero is CAR,
    (m2c mwrite)          ; write it,
    (vc0 snormal)         ; We have to pop value stack too,
    (vc1 spopl)           ; pop one,
    (vc2 swidle)          ; no writes,
    (vc3 spwrite)         ; save new stackpointer.
  )
)
(defun app-res-vs-rplacd ()
  (print-2-cr "Doing a RPLACD on VS" " ")
  (appres text-vs-rplacd-1)
)
(setq text-vs-rplacd-1
  '(
    (comment "RPLACDing Vunder with Vtop")
    (comment "Since you can *only* rplacd conses, use M2")
    (m2v0 b-vd0)          ; stacktop is value to poke,
    (m2v1 b-vd1)          ; undertop is where to poke it,
    (m2v2 1)              ; offset zero is CDR,
    (m2c mwrite)          ; write it,
    (vc0 snormal)         ; We have to pop value stack too,
    (vc1 spopl)           ; pop one,
    (vc2 swidle)          ; no writes,
    (vc3 spwrite)         ; save new stackpointer.
  )
)

```

Appendix B

DIS Instruction Summary

DIS Instruction Summary

Dedicated Output Busses Available

<i>Name</i>	<i>Mnemonic</i>	<i>MC Usage</i>
M0 Output	B-MD0	Instructions
M1 Output	B-MD1	Symbol Table
M2 Output	B-MD2	Cons Cells
Value Stk Top	B-VD0	Top Argument
Value Undertop	B-VD1	Below Top
Value Stkpointer	B-VSP	Current SP
Pending Stk Top	B-PD0	Entry Ptr
Pending Undertop	B-PD1	Arg Frame Ptr
Return Stk Top	B-RD0	Return Loc
Return Undertop	B-RD1	Result Ptr
Binding Stk Top	B-BD0	Rebound Value
Binding Undertop	B-BD1	Rebound Addr
ALU output	B-AD0	ALU data
Seq Next Addr	B-SD0	Next Addr
Network Output	B-ND0	Networking
Literal Data	B-CMI	Immed Data

ALU Control

AV0	ALU datum A (bus)
AV1	ALU datum B (bus)
AC0	Boolean op (0 to 15)
AC1	Data Selector selzero sel1 sel-1 selAV0
AC2	Format select fmtfix fmtfloat
AC3	Combinational combadd combsubtract combmultiply combddivide
AC4	Shift Count (0-255)

DIS Instruction Summary

Overall Instruction Format

A *DIS* assembly language program is a LISP-style list of instructions. A single *DIS* instruction usually contains multiple fields. The *DIS* linker will assemble the instructions and resolve location references at load time.

Example:

```
((inst1)
 (inst2)           ; a DIS program.
 (inst3))          ; three instructions long
```

Fields within Instructions

	SEQ	Net	ALU	Bind	Retn	Pend	Value	M2	M1	M0
--	-----	-----	-----	------	------	------	-------	----	----	----

bit 255 Each unit in a *DIS* processor can cycle on each instruction. Units always read bus data output on previous cycle except for immediate data, which is from the current cycle. IDLE units do NOT change their outputs. bit 0

Example of a *single* instruction (one cycle to execute):

```
((comment "Push an immediate integer onto Value Stk")
 (location push-integer) (cmi 42)
 (m0c0 mread) (m0v0 b-sd0) (sv0 b-md1) (sv1 b-sd0)
 (sc0 1) (vd0 b-cmi) (vc0 snormal) (vc1 push1)
 (vc2 swtop) (vc3 spwrite))
```

Assembler/Optimizer Pseudo-ops

```
(comment "Anything you want") ; a comment
(location location-name)      ; a named location (global)
(crushmax n) ; inhibits optimizer motion more than n forward
(reserve unit unit...) ; reserves units for later use
(cmi n) ; places integer n on immediate data bus
(cv1 x) ; places x in pointer-count field of immediate data.
(cv2 y) ; places y in object-count field of immediate data.
```

***DIS* Instruction Summary**

Memory Control

n = 0, 1, or 2

MnV0 Mem Addr In (bus)
MnV1 Mem Data In (bus)
MnC0 Control
 Midle
 Mread
 Mwrite
 Mfadd
MnC1 Addr Index (0-7)

Stack Control

s = V, P, R, B

sD0 Top Input (bus)
sD1 Below Input (bus)
 or Subhead SP
sC0 First Control:
 Sidle
 Snormal
 Subhead
sC1 Stkptr Motion:
 (-8 to +7)
sC2 Write Action:
 Swidle
 Swtop
 Swt-1
 Swboth
sC3 Save new SP:
 SPidle
 SPwrite

Sequencer Control

SV0 Next Instr (bus)
SV1 Primary Next Addr (bus)
SV2 Secondary Next Addr (bus)
SC0 Primary increment (0 to 3)
SC1 Secondary increment (0 to 3)
SC2 Condition Code Mask
 SEQMASKZERO
 SEQMASKLESS
 SEQMASKNIL