A Pascal Machine Architecture Implemented in Bristle Blocks, a Prototype Silicon Compiler

Technical Report 2883

Thesis by: Larry Seiler
Advisor: J. Craig Mudge

In Partial Fulfillment of the Requirements
for the Degree of Master of Science

California Institute of Technology
Pasadena, California

1980
(Submitted July 3, 1979)

Abstract

This thesis presents the multi-chip design of an architecture which directly implements the language Pascal. The design uses custom VLSI rather than standard chips in order to increase speed and reduce the number of chips needed.

The integrated circuits comprising the architecture are designed using Bristle Blocks, a chip design tool developed at Caltech by Dave Johannsen.[6] Bristle Blocks is called a *silicon compiler because it will put together an entire integrated circuit from a high level description* of its function. Bristle Blocks can be used to design datapath processor chips, where external microcode is used to control operations on data busses inside the chip.

The Pascal machine architecture presented here is based on the EM-1 instruction set designed by Andrew Tannenbaum.[11,13] The EM-1 instruction set is intended to allow efficient compilation of stack-based, high level languages. Tannenbaum supplies static frequency data which is used heavily in making design decisions in the Pascal machine architecture.

VLSI design has several important differences from design using standard components. A large amount of function can be placed on a single chip, e.g., approximately 30,000 transistors on the Intel 8086, but only a small number of pins are available for off-chip communication (typically 64 or less). This requires designs to be highly modular. In the NMOS technology used at Caltech, driving signals off-chip takes up to ten times the time and energy of on-chip communication. This requires inter-chip communication to be limited as much as possible. Finally, the large amount of computing power available in VLSI encourages the use of concurrency to gain execution speed.

This thesis is structured as follows. The thesis begins with a section defining the principles to be followed in designing the Pascal system architecture. Following that are sections describing Bristle Blocks and the EM-1 architecture. Next, the overall architecture of the Pascal machine is described, followed by sections detailing the system data busses, the common elements in the processors which make up the system, and the processors themselves. A conclusion section summarizes the work, provides a brief critique of Bristle Blocks, and includes recommendations for further work. Finally, the appendices document the Bristle Blocks datapath elements and the EM-1 instruction set.

Table of Contents

Figures

Tables

1. Design Principles

In selecting the principles to follow in designing the Pascal architecture, it is important to keep in mind the differences between designing in custom VLSI and designing using standard SSI or MSI parts. Many of the assumptions behind traditional switching theory are false when applied to VLSI.[7] For example, switching theory teaches how to minimize logic elements, but ignores the communications paths between the logic elements. In VLSI, it is the wires that take up most of the chip area and it is the delay involved in driving them which takes up most of the time. So in VLSI, communication costs are more important than logic costs.

When sending information off-chip, communication costs are even more important. Using conservative processing technologies, driving signals off-chip can take up to 10 times the time and energy as on-chip communication. With design rules shrinking as the technology advances, that factor will worsen to 25. Therefore, multichip VLSI systems must be designed to minimize interchip communication.

Another difference is the issue of modularity. In a circuit designed with SSI, it is easy to test internal signals to find a fault. Using VLSI, is is very difficult to gain access to internal signals. This means that VLSI chips must be designed with well-defined interface characteristics. Given the large amount of processing power available on each chip, it is acceptable to design them such that they check their inputs to insure that the interface protocol is being followed. If an error condition developes, the VLSI chip should be able to detect it and report it.

Given that interchip communication is at a premium, it is reasonable to have the separate VLSI chips operate as independently as possible. Requiring interchip communication to use asynchronous, request/acknowledge protocols increases that independence. Given the large variety of ways in which the various chips can communicate, it is also acceptable to require interchip communication to be based on a message protocol. After the design is fairly fixed it would be possible to optimize the communication busses. At the beginning of a design task it is important to include all the flexibility possible.

The result of the above principles is a multi-chip architecture where each chip is a custom designed VLSI processor. This approach is seldom used in the computer industry because the cost of hand designing several VLSI chips for one project is prohibitive. Standard chip sets, limited custom design, and logic arrays are typically used instead to gain some of the advantages of VLSI at a reasonable cost. None of these methods are completely satisfactory for designing an architecture according to the above principles. Instead, we propose the use of a silicon compiler. The advantages of using the Bristle Blocks silicon compiler are that it produces powerful, customized processor chips and greatly reduces chip design time. Without Bristle Blocks or some other silicon compiler, the construction of the architecture presented in this thesis would not be feasable.

2. Bristle Blocks Description

Bristle Blocks is an integrated circuit design tool developed at Caltech by Dave Johannsen.[5] The name refers to the design style which Bristle Blocks enforces. Users select or define low level cells, or blocks, which are specified by their internal structure and their interface points, or bristles. Blocks are composed to make larger blocks, with the interconnect work done completely by the Bristle Blocks system. Bristle Blocks was programmed in ICL (Integrated Circuit Language), a language developed at Caltech by Ron Ayres.[1,2]

Drawing an analogy between integrated circuit design and software design, Bristle Blocks is referred to as a silicon compiler. In Bristle Blocks, the designer specifies the blocks and their interconnect structure in a high level language. Designing ICs by laying down individual wires and boxes corresponds to programming in machine code. A simple artwork language which supports macros (symbols) corresponds to using an assembler. Bristle Blocks qualifies as a compiler because the blocks it manipulates are not just fixed structures, whose exact positions and interconnect are specified by the designer. Instead, the blocks are parameterized functions whose placement and interconnect are done by the Bristle Blocks system. The blocks are parameterized so that Bristle Blocks can adapt a single block to a variety of conditions. For example, blocks can be stretched to make the pitch of their bristles match up with those of adjacent blocks. Also, a block can be specified so that the ratios of its bus drivers change depending on the capacitance of the bus it is driving.

2.1 Design Constraints

Bristle Blocks is not intended to be used to design all possible integrated circuits. Instead, it is designed to build chips with a very specific architecture. Other silicon compilers will be developed to handle other architectures. For example, an IC design system which uses clocked logic equations to specify a heirarchy of PLAs is presently being developed at Caltech by Ron Ayres.[3]

The present version of Bristle Blocks implements a datapath processor driven from an external microcode store. The data chip used in the LSI-11 chip set is an example of such a processor. The Bristle Blocks chip is driven by a two phase, non-overlapping clock. The main section of the chip is the datapath, which contains two precharged data busses. Memory registers, shifters, ALUs, and I/O ports are some of the blocks that can be included in the datapath section. The control signals from the datapath are generated in the decoder section, which is essentially the AND plane of a PLA. The decoder inputs microcode bits from off-chip and performs logic functions on them. Some signals from the datapath, such as the carry out bit from an adder, are also used by the decoder. The decoder outputs its functions to a buffer section, which latches them and drives them onto the datapath. The buffer section also feeds signals from the datapath section to the decoder. Figure 1 shows the physical arrangement of these sections on the chip. The widths of the datapath, buffer, and decoder sections are matched by the Bristle Blocks system. The pads are spaced evenly around the perimeter of the chip.

The datapath section contains data busses and power busses, which run horizontally in metal. Control lines generally run vertically in polysilicon. Figure 2 shows the arrangement of busses for each bit in the datapath. A ground bus runs through the center of the cell and VDD busses run at top and bottom. The corresponding bits of the two data busses run above and below the ground bus. Blocks designed for the datapath must fit into this structure.

Datapath elements must be defined in such a way that they fit together well. The rules for the left and right edges of a datapath element are fairly easy. All features of the element must be one half of their respective design rule width within the boundary. An exception is made for metal to diffusion contacts. They may be placed on the four points defined by the intersections of the data busses and the cell boundaries. These contacts are permitted because they can be shared by adjacent datapath elements and many datapath elements need them.

The rules for the vertical spacings of the datapath cells are more difficult. To simplify the interconnect along the datapath, all datapath blocks are required to have the same spacings between each of their horizontal busses. Each datapath block is defined to be stretchable, that is, they are defined such that the spacings between the busses can be increased arbitrarily beyond a certain minimum spacing. When the datapath is generated, Bristle Blocks simply finds the maximum required spacings, and stretches out all of the datapath blocks to match. This is done by defining the internal features of the block relative to the global variables Y1 to Y4, whose values correspond to the centers of the data and VDD busses. The center of the ground bus is at zero.

2.2 Cell Library

Bristle Blocks places its emphasis on automating the high level aspects of chip design and, at least for now, provides little assistance in the job of laying out datapath blocks. There are several reasons for this. For a human designer, the bookkeeping tasks associated with high level layout and interconnect of a chip is the most tedious and error-prone part of the job. A computer program can do the job more efficiently and more reliably. On the other hand, efficient small cell design is much easier and less error prone for the human designer. The most telling reason, though, is that there has not yet been sufficient time to develope low level design aids for Bristle Blocks.

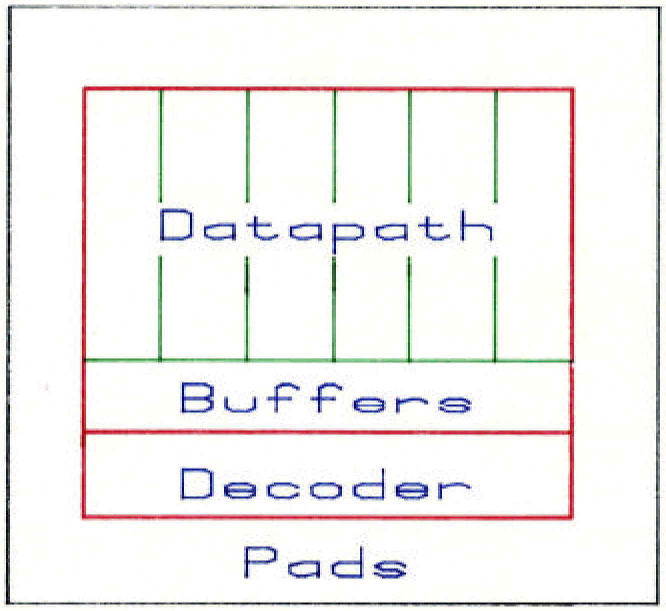


Figure 1: Physical Structure of a Bristle Blocks Chip

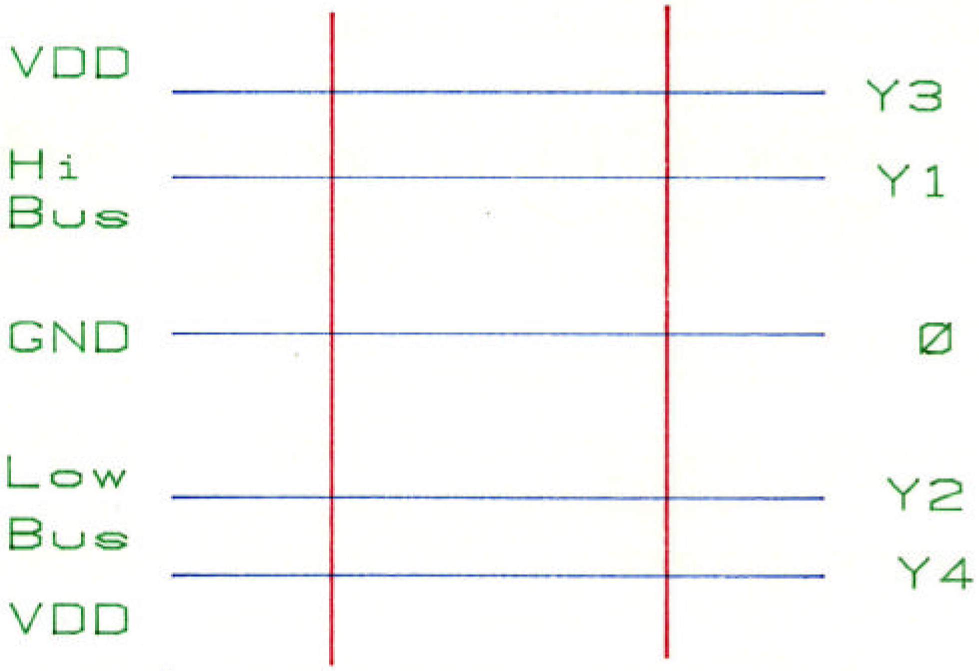


Figure 2: Bus Structure of a Datapath Bit Cell

If each user had to personally design all the datapath elements he wanted, Bristle Blocks would not be very useable. However, every datapath element that has been designed by any user can be entered into a cell library and used by any other user. For example, many of the datapath elements used in the Pascal system were designed by Dave Johannsen as part of his use of Bristle Blocks in designing a recent version of OM 2, a 16 bit microprocessor datachip. He in turn is now able to use the datapath elements designed by the author for the Pascal system. The cells designed for the Pascal system are documented in Appendix A.

The cell library system works as follows. A datapath element is defined by three code files and one data file. The code files are the runtime function file, which is executed when a chip is generated, the source file, which defines the internal structure of the cell, and the details file, which contains information used by both the runtime and source files. The source file is compiled to produce the data file, which is used when the chip is generated.

The layout is not the only information that can be recorded in the cell library. Bristle Blocks allows the use of alternate representations, which are different representations of the same cell. The alternate representations used presently are the layout, sticks representation, transistor diagram, logic diagram, and block diagram. The block diagram abstracts the datapath elements into a row of labeled rectangles, with the data busses above and below. The data busses are connected to the datapath blocks; arrows indicate whether the bus is read or written by that block. Having a number of representations available is a great aid to documentation and provides the possibility for verifying the correspondence of physical and logical representations.

2.3 Processor Types

Many types of processors can be generated using the Bristle Blocks chip architecture. Figure 3 illustrates some possibilities relevant to the Pascal system. The multi-chip processor uses two Bristle Blocks chips: one to process data and one to act as microcode sequencer. The single chip processor combines the two into one chip. The memory interface processor is a useful as an intelligent memory controller. Finally, the one instruction computer produces its own microcode bits internally and accesses an external memory for datapath transfer instructions.

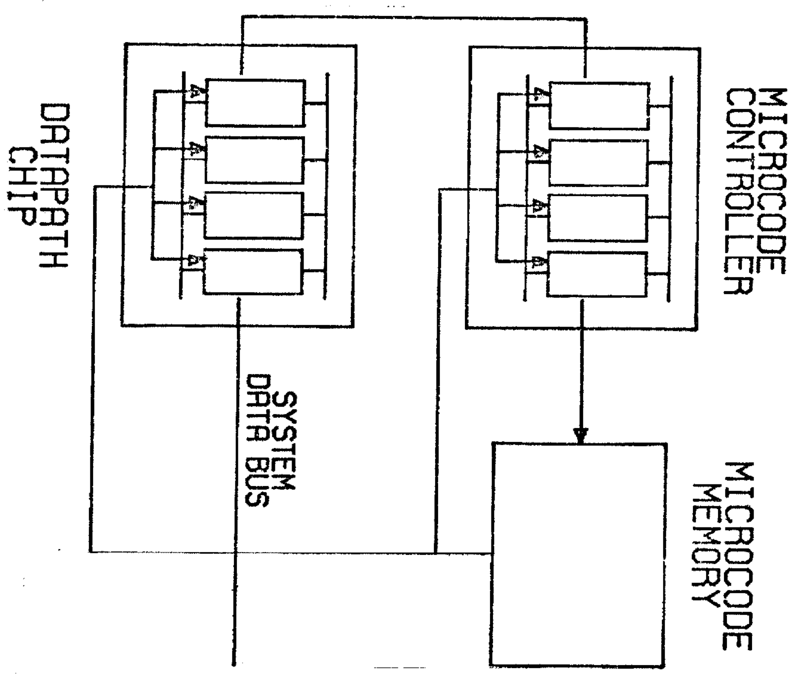


Figure 3a: Multichip Processor

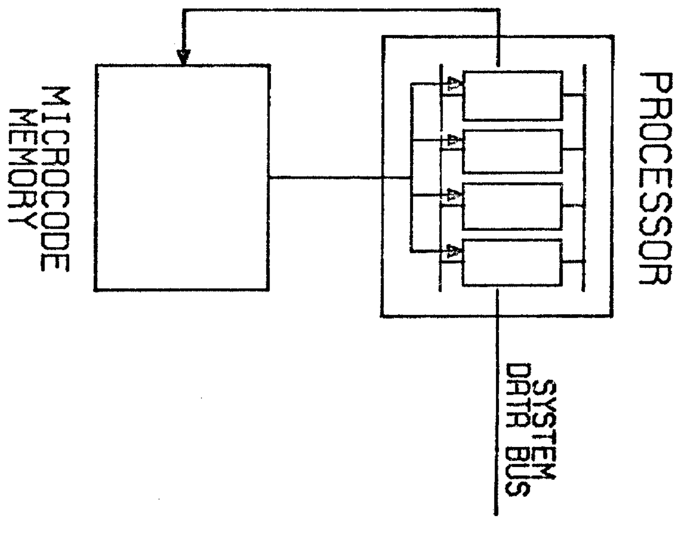


Figure 3b: Single Chip Processor

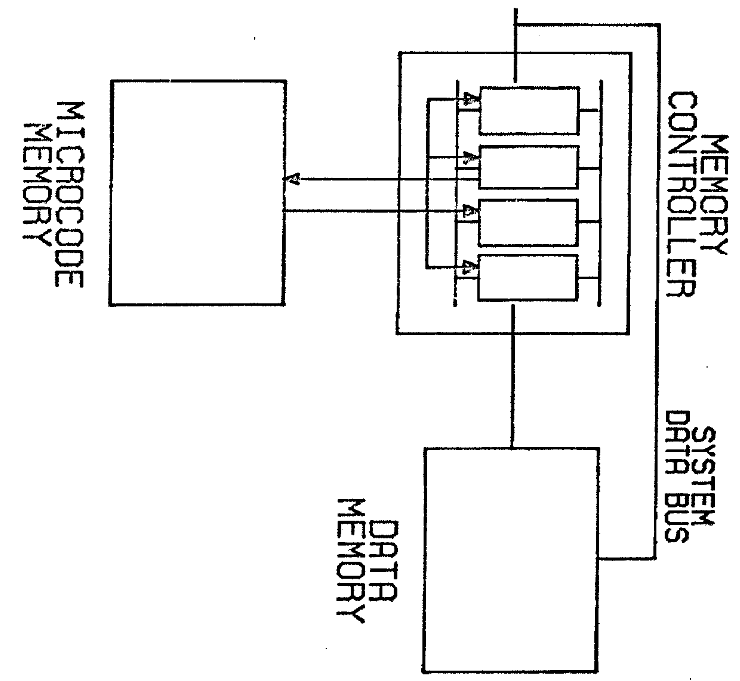


Figure 3c: Memory Interface Processor

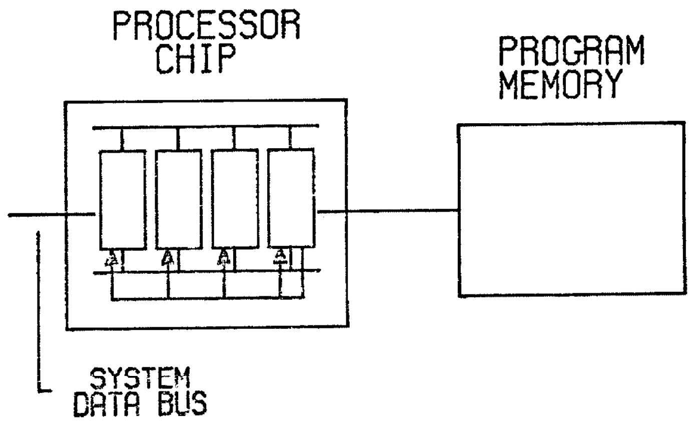


Figure 3d: One Instruction Computer

3. EM-1 Instruction Architecture

Before defining the Pascal system architecture, it is necessary to describe Tannenbaum's EM-1 architecture. EM-1 is an experimental machine architecture designed to be implemented on microprogrammed machines. It is intended to be used to execute stack-based, block structured languages. The EM-1 instruction set is designed to be compact and easily interpreted. It is also intended to simplify code generation by high level language compilers. The information in this section is taken from a paper by Tannenbaum, Stevenson, and van Staveren.[13]

3.1 EM-1 Machine

The EM-1 is similar in some ways to the PDP-11 series. It manipulates 16 bit data words. Addresses are 16 bits long and the least significant address bit selects between the high and low byte of a word. If 16 address bits are not sufficient, a virtual address upgrade can be made by adding an 8 bit segment number to each physical address. Address variables in memory and on the stack then take two words rather than one. This change is not difficult to make because physical addresses do not occur in EM-1 code, just relative displacements. Also, only a small number of instructions manipulate addresses on the stack.

The EM-1 architecture defines a set of machine registers which, along with the runtime stack, define the state of the machine. The registers and their functions are given in table 1 below.

Register	Function
Program Status Word	Saves program status and priority
Procedure Base	Points to the start of the code area
Program Counter	Points to the next instruction to execute
Procedure Descriptors	Points to the procedure descriptor table
External Base	Points to base of stack; used for external variable access
Local Base	Points to the zeroth local variable in the current procedure
Stack Pointer	Points to highest occupied word on the stack
Heap pointer	Points to lowest occupied word on the heap
Memory Limit	Highest byte of memory available in the implementation

Table 1: EM-1 Machine Registers

3.2 Instruction Set Format

The EM-1 instruction set is defined in appendix B. The instructions are described in section 1. Sections 2 to 4 give static instruction frequencies and other information used later in the thesis. Section 5 gives more detailed frequency data.

EM-1 instructions are specified to be multiples of one 8 bit byte in length. Their main difference from a standard instruction set is that they do not contain subfields: operands either take up an entire byte, or the operand is encoded into the opcode. For example, the load local variable instruction (lol x) is encoded as an opcode followed by an offset from the local base register. Since accesses to the first few locals are very common, there are bytecodes set aside for loading the zeroth through seventh local variables. These instructions would be executed by separate microcode routines. Code space is reduced because the most common load instructions do not require operand bytes. In choosing such instructions, the instruction set designer is trading off code compaction against microcode size. The greater the number of optimized bytecodes, the greater the number of microcode routines needed to execute them.[10]

Table 2 below describes the instruction formats used in EM-1. There are two 1-byte instruction formats. One is for instructions whose operand is coded into the opcode, as with the load local variable instruction mentioned above. The other is for instructions which take no operand. There are also two 2-byte instruction formats. Format f2a is for instructions with short offsets. Format f2b is for zero operand instructions that are not common enough to use format f1b. They are referred to as escaped instructions. The escape code has a value of zero. The two 3-byte formats are for normal instructions with a large offset, and escaped instructions with a short offset. Finally, there is a 4-byte format for escaped instructions which have a large offset.

Format	First Byte	Second Byte	Third Byte	Fourth Byte
f1a	opcode + offset			
f1b	opcode			
f2a	opcode	offset byte		
f2b	escape	opcode		
f3a	opcode	offset high byte	offset low byte	
f3b	escape	opcode	offset byte	
f4a	escape	opcode	offset high byte	offset low byte

Table 2: EM-1 Instruction Formats

An important advantage of the EM-1 architecture is that the specific mapping of the instruction set onto bytecodes is not a fixed part of the architecture. The mapping and the selection of optimized bytecodes can be specified by the user to fit his application. This is accomplished by having compilers output a compact form of assembly code. An assembler then translates this to the actual bytecodes, choosing the most efficient instruction formats available and performing local optimizations. This greatly simplifies compiler code generation. It also means that, in an application such as this, the bytecode mapping can be chosen to match the architecture, rather than trying to match the architecture to an arbitrarily defined mapping.

3.3 Procedure Call and Return Mechanism

A procedure call is accomplished in the EM-1 architecture using three separate steps. First, a mark stack instruction (mrk n) is executed. This creates a procedure administration area on the stack. The administration area contains the static and dynamic links. It also contains a space for the return address, which is filled in later. Next, instructions are executed which load the actual parameters onto the stack. Finally, the procedure call instruction (cal n) is executed. This instruction reads in procedure descriptor *n*, which gives the procedure starting address and its number of words of parameters. The latter value is subtracted from the stack pointer to find the address of the zeroth actual parameter. This address is loaded into the local base register and is also used to find the return address location in the administration area. The present PC value is saved, the procedure starting address is loaded into the PC, and execution continues within the called procedure.

A procedure return instruction (ret n) undoes all this. The local base register is used to find the administration area so that the PC can be loaded from the return address and the local base register can be loaded from the dynamic link. The stack is then trimmed down past the administration area. After the stack is trimmed back, the *n* words from the old top of stack are loaded onto the stack. This allows a procedure to return a value.

3.4 Alternate Context

Tannenbaum provides a mechanism to reduce the code space by reducing most procedure call instructions to a single byte. This is done by defining an alternate instruction context, in which some of the bytecodes are used as single byte procedure call instructions. To use this method, the bytecode mapping must be chosen such that all instructions which are commonly used to load actual parameters have bytecodes in one group and all instructions which are not commonly used to load actual parameters have bytecodes in another group.

The two groups are specified to the assembler. Whenever it finds a mark stack instruction, the assembler checks to see if any instructions from the uncommon group occur before the next procedure call instruction. If not, the assembler issues a special mark stack instruction which sets an alternate context flag. The entire uncommon section of the bytecode space is now free to be used for optimized procedure call bytecodes. If the descriptor number of the called procedure is less than the number of uncommon instruction bytecodes, a single byte can specify the procedure call. The alternate context flag is cleared by the procedure call instruction. In Tannenbaum's bytecode mapping, 122 bytecodes (nearly half of the bytecodes) are specified to be uncommon.

3.5 Array Access

In the EM-1 machine, arrays are implemented using descriptors. There is one descriptor for each array type declaration. An array descriptor contains information specifying the lower bound, upper bound, and array element size.

An array element is loaded onto the stack in the following way. First, the starting address of the array is loaded using an instruction such as load address external (loe x). Next, the array index is pushed. Finally, a load array element instruction (lar x) is executed, where x is the external offset to the array descriptor. The load array element instruction reads in the array descriptor and compares the index to the upper and lower bounds. If it is out of range, a runtime error is generated. Otherwise, the specified array element is loaded onto the stack.

4. Pascal System Architecture

The major issue involved in designing the architecture of the Pascal system is partitioning of function between the chips. The task of executing the EM-1 bytecodes must be partitioned in such a way that each chip has its own clearly defined function and requires a minimal amount of communication with the other chips. To be efficient, the partitioning must allow a large degree of concurrency. For each aspect of the design, the limiting factor must be recognised and dealt with.

4.1 Bus Structure

The first aspect to consider is the bus structure of the system. At the highest level, the system looks like a single central processor communicating with main memory. To this extent, the Pascal system architecture is constrained to match standard architectures. However, the CPU of the Pascal system does not need to be a single processor. Considerations of concurrency imply that it should be split up into several different processors. This separation could naturally occur along the lines of the different types of accesses performed on main memory.

In the language Pascal, three basic types of memory access are done. First, there are the instruction fetches. Pascal code is re-entrant, so instructions can be fetched independently from any other memory accesses occurring on the bus. That is, there are no conditions placed on the relative orders of the instruction fetches and the other memory accesses. Second, there are stack operations. Accesses to the evaluation stack are also independent from the other memory accesses. Finally, there are the variable accesses. As with the other two, the order of the variable accesses is essentially independent of the other memory accesses.

These three types of memory access can be done by three different processor chips which communicate with each other over a second message bus. The three processors will be called the instruction unit, the memory manager, and the stack manager. The instruction unit issues instructions over this bus to the other processors, which execute them.

4.2 Local Stacks

Separating the three kinds of memory access increases concurrency, but does not affect the memory bandwidth problem, which is the major bottleneck in most computer systems. Therefore we must look for ways to eliminate accesses to main memory. Instruction fetches could be reduced whenever a short loop is encountered by saving all the instructions from the loop inside the instruction unit. Variable accesses could be reduced by putting a cache within the memory manager. However, the greatest benefit would be realized if stack accesses could be eliminated. This is because the stack must be accessed by each processor on almost every instruction.

Suppose that each of the processor chips in the Pascal system contained a local stack, that is, a set of registers internal to the processor which would contain the top few stack words. These registers could be pushed and popped without having to access main memory, provided that the evaluation stack stays short. Frequency data compiled by Tannenbaum[12] indicates that the evaluation stack will usually be very short. The local stacks would be kept up to date by messages passing over the second message bus, hereafter referred to as the stack bus. The stack manager would have the job of interfacing the local stacks to main memory.

At this point we must ask how much we have gained by the introduction of local stacks. They certainly eliminate many main memory accesses, but they add equally many messages to the stack bus. Every time the memory manager, for example, wants to pop a word from its local stack, it must send a pop stack message across the stack bus, so that all the local stacks pop a word.

The local stacks do not reduce the number of messages that are passed, but there are several important points in their favor. First, they increase the modularity of the system. Only the stack manager needs to know about the stack pointer and the stack in main memory. It would be difficult for several independent processors, each with its own copy of the stack pointer, to stay synchronized. Second, the local stacks remove stack accesses from the critical path. For a stack in main memory, a processor must calculate the address, arbitrate for the bus, and perform the read before continuing execution. With a local stack, the word can be read immediately, and execution can go on concurrently with the arbitration for sending the pop stack message over the stack bus. Finally, the stack bus is faster than the memory bus. The memory bus must connect components on several circuit boards, possibly in more than one cabinet. Since the stack bus connects only a small number of chips, it can easily be restricted to a single circuit board.

4.3 System Structure

So far we have defined a CPU with three processors, each of which communicates with both the memory bus and the stack bus. Many of the EM-1 instructions, such as the arithmetic instructions, only use the stack and do not require any communication with main memory. If we add integer and floating point processor chips, we get a system that looks like figure 4. The instruction unit fetches instructions from main memory, decodes them, and issues them over the stack bus to the other processors. The memory manager executes all instructions that access variables in main memory. The stack manager keeps track of the local stacks, and also executes those instructions which require accessing the stack in main memory. The integer processor executes the integer arithmetic instructions plus some boolean instructions, and the floating point processor executes the floating point instructions.

It should be noted at this point that each processor is free to execute its instructions in any way it wishes. For example, the memory manager could contain a standard cache, or it could contain registers for the first few local variables. The instruction unit could save short loops internally, prefetch from both sides of conditional branches, or use other instruction prefetch strategies. The specific method chosen by a particular processor affects the speed of the total system, but it does not affect the correct functioning of the other processors, because all interactions between processors use an asynchronous request/acknowledge protocol. This is the vitally important modularity condition, that a module may operate in any way that satisfies its interface conditions without affecting the correct functioning of the rest of the system.

4.4 Instruction Issue

Instruction issue must be done using a request/acknowledge protocol. More than this, the instruction issue algorithm must be designed to minimize the amount of imformation the instruction unit has about the rest of the system. In a typical computer system, the instruction dispatcher controls which execution unit executes which instruction. With our design principles, a more distributed system of control is desirable. The instruction unit sends out an instruction on the stack bus. If one of the processors recognizes it as an executable instruction, it sends a message back to the instuction unit indicating that it has taken the instruction. If several processors recognize the instruction, the first to get control of the bus sends the taken message and executes the instruction. If no taken message appears within a certain amount of time, the instruction unit re-issues the same instruction. This permits processors to ignore instructions being passed across the bus when they ae busy doing something else. If the instruction unit issues the same instruction many times in succession without a response, it assumes that either the stack bus or one of the other processors has failed.

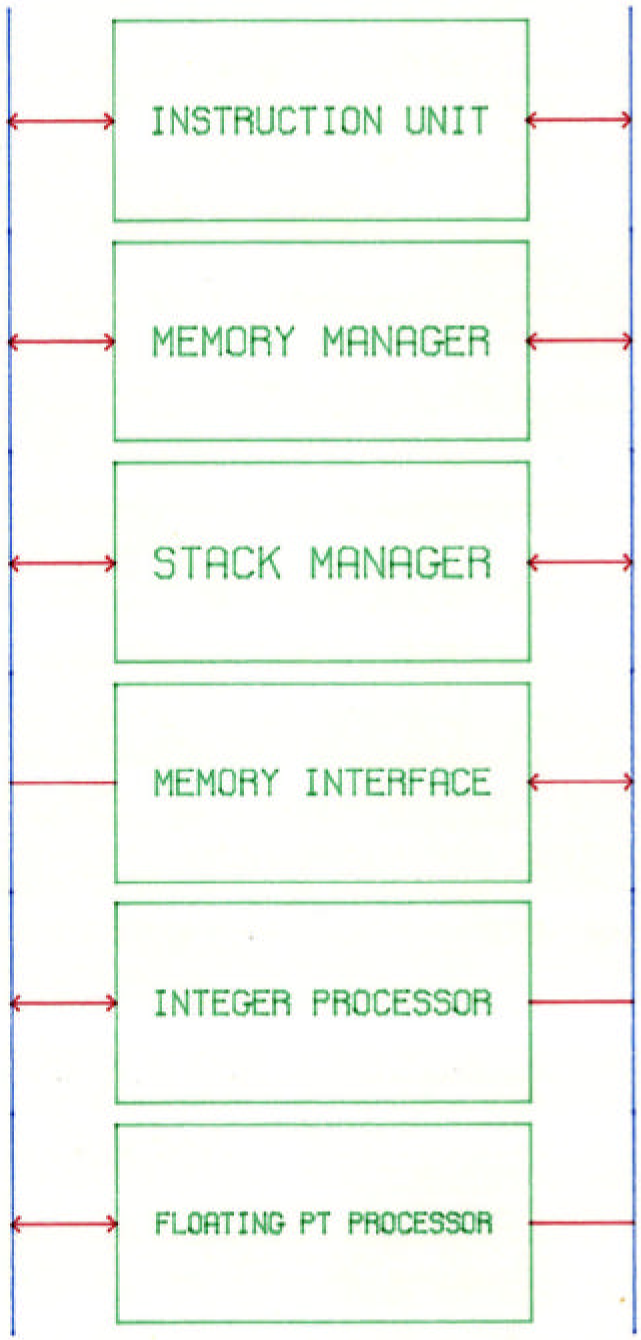


Figure 4: Pascal System Architecture

5. Bus Definitions

This section defines the logical function of the two data busses used in the Pascal system: the memory bus and the stack bus. The physical properties of the busses are not specified beyond the fact of their being message busses. As message busses, they require asynchronous data transfer and an asynchronous method of arbitrating for bus control. One message bus that could be used is the triMOSbus, developed at Caltech.[9,11] A triMOSbus interface and chip test system has been developed at Caltech by Greg Efland for his Master's Thesis.[4]

5.1 Memory Bus

The memory bus is used for main memory access by the instruction unit, the memory manager and the stack manager. Each of the three has slightly different requirements. The instruction unit does block reads almost exclusively. Since it is prefetching instructions, the memory access time is noncritical. Therefore, the instruction unit should have the lowest priority in arbitrating for control of the memory bus. The stack manager does mostly block reads and writes along with a few alternating reads. An alternating read is used to do operations such as string comparison: single words are read alternately from the two multiword operands. Its accesses are sometimes on the critical path, so the stack manager should have a higher priority than the instruction unit. The memory manager does mostly single word loads and stores, but also does some block reads, block writes, and block moves. Since operations done by the memory manager are often on the critical path, it should have the highest priority in memory access.

In the present version of the Pascal system only two messages are defined for the memory bus. They are block read and block write. Consulting the instruction frequency information in appendix B, it can be seen that block moves and alternating reads are not done very often. Adding up the static frequencies of instructions that would use an alternating read message gives 0.25%. The static frequencies of instructions that would use a block move message sum to 1.06%. It would be easy to add these and other messages if the need were later thought to justify them or if another language implemented with this architecture used them.

Having determined the messages which are used on the memory bus, it is necessary to decide the width of the memory bus. The EM-1 architecture uses 16 bit data and 16 bit addresses, so it is very natural to use a 16 bit bus. A memory bus message would consist of a message header word, followed by the address word, followed by a specified number of data words. The number of data words and the message type would be specified in the message header word. For a block write the data words would be provided by the processor controlling the bus; for a block read they would be driven onto the bus by whatever memory unit recognizes the message address address as being in its range. Because there are only two messages defined, the header word will contain many unused bits. In a future system with virtual memory these bits could be used to contain the segment number.

5.2 Stack Bus

The stack bus is used for issuing instructions, pushing and popping the local stacks, and sending certain error and reset messages. Table 3 gives the complete set of messages which need to be sent over the stack bus. The instruction unit uses the OPCODE and ESCAPE OPCODE messages to issue instructions to the execution units. The RESTORE STACK message is issued by the stack manager. Its function is to rewrite the local stacks after a RESET message or after a STACK UNDERFLOW message has indicated that an execution unit needs data that was pushed from the bottom of its local stack.

Message	Description
Opcode	An EM-1 opcode with 0, 1 or 2 bytes of data
Escape Opcode	An EM-1 escaped opcode with 0, 1 or 2 bytes of data
Accept Instr	Issued by chip which will execute the previously issued opcode
Pop Stack N	Pop n words from local stack, push following data onto stack
Stack Underflow	Issued by a chip when its local stack has lost data
Restore Stack	Issued by stack manager in response to above
Runtime Error	Informs instr unit of a runtime error (followed by an error code)
Interrupt	Informs instr unit of an interrupt (followed by an interrupt code)
Accept Interrupt	Informs interrupting device that the interrupt has been accepted
Reset	Resets the system to initial state

Table 3: Stack Bus Messages

The next question to be resolved is the width of the stack bus. It could be made 16 bits wide, the same as the memory bus, but it seems best to limit the stack bus to 8 bits. The reasons for this are as follows. The EM-1 opcodes were specifically chosen so that most instructions would be only one byte long. ACCEPT INSTR and other messages also need be only one byte long. The only place where a larger bus would pay off is in the POP STACK N message, where the data would take twice as long to transfer with an 8 bit bus, although arbitrating for the bus and transmitting the message header would take the same time. Therefore if it is common for instructions to push several data words onto the stack, a 16 bit bus would be preferable. However, the table of mnemonic frequencies in section 3 of appendix B indicates that the average number of words pushed onto the stack by each instruction is ~1. Therefore the best data width for the stack bus is 8 bits rather than 16.

In the above discussion, it was implicitly assumed that an instruction issue message (ie OPCODE or ESCAPE OPCODE) consists of the bytes that represent the instruction in main memory, with no translation done by the instruction unit. Since the representation in main memory was chosen for compactness and ease of decoding, it is reasonable to do this. The byte codes assigned to the other stack bus messages can be chosen from the set of opcodes executed within the instruction unit, since these are never issued onto the stack bus. This provides enough bytecodes so that POP STACK N can be given 16 separate codes, corresponding to popping 0 to 15 words from the stack. There should also be a long form of the POP STACK N message, which represents n as a two byte number following the message header. This long form will be very seldom used and need only be recognized by the stack manager, which will respond by adjusting the stack pointer and issuing a RESTORE STACK message.

6. Processor Section Definitions

There are several functions which are performed by more than one of the processors in the Pascal system. This section gives functional definitions for the datapath sections which are used to implement these functions. These processor chip sections are built out of the primitive datapath elements defined in appendix A and some other datapath elements designed by Dave Johannsen for the Bristle Blocks cell library.

6.1 Stack Bus Interface

Each chip connected to the stack bus must have a stack bus interface. This unit interfaces the synchronous processor to the asynchronous stack bus, and formats data passed between the processor and the message bus. Data formatting is necessary because the stack bus is 8 bits wide and the chip datapath is 16 bits wide.

Data formatting is handled by a byte manipulator contained in the stack bus interface. Three types of formatting are done on input and two are done on output. When reading in data, the next input byte always appears as the low order byte of the word. The high order byte may be zero, a duplicate of the present input byte, or the previous input byte. A full word may be written to the byte manipulator to be output high byte first onto the stack bus. Alternately, a word may be written of which only the high order byte is output.

In the present system, the stack bus interface is mostly implemented off-chip. The on-chip section of the interface consists of a standard I/O register which can drive its contents off-chip or latch data from off-chip. There are several reasons for not including the stack bus interface on-chip in this version of the Pascal architecture. First, the interface cannot be designed because the stack bus itself is not completely specified. Second, it would be difficult to design the asynchronous interface and sequencing circuit in the present version of Bristle Blocks. Finally, even if the above problems were solved, the time required to design an on-chip stack bus interface would be too great to include it in this version of the Pascal system.

6.2 Local Stack Controller

The factor which makes the processor chips efficient is the local stacks contained in each. The execution section of each chip is free to read values out of the local stack, but the state of the local stack can only be changed by messages received from the stack bus. Since all the local stacks receive each message, all the local stacks are always in the same state.

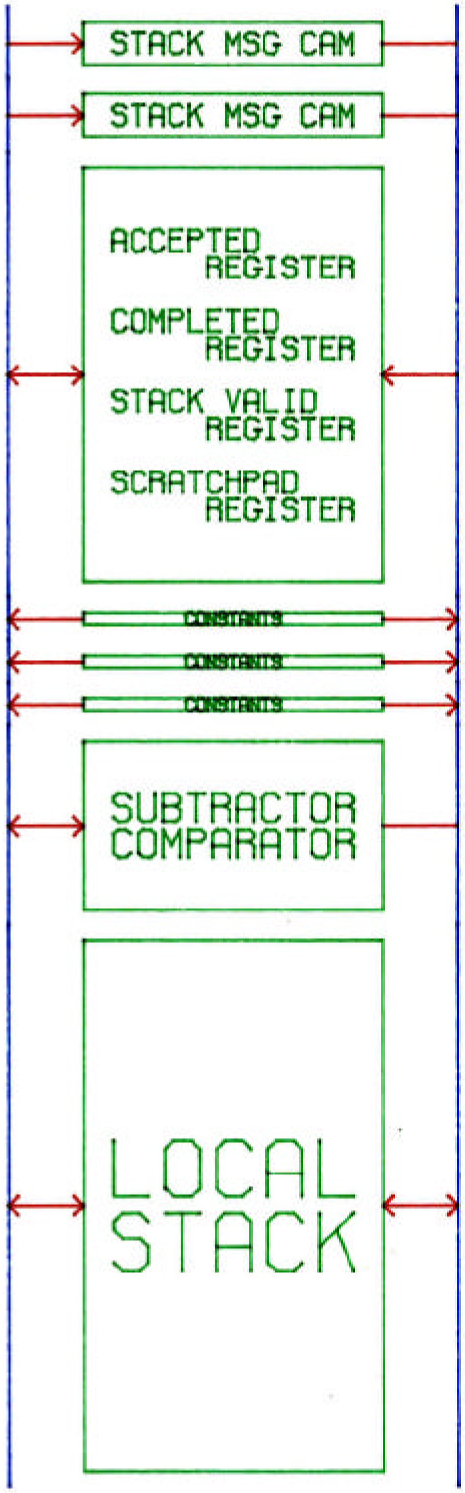


Figure 5a: Local Stack Controller Block Diagram

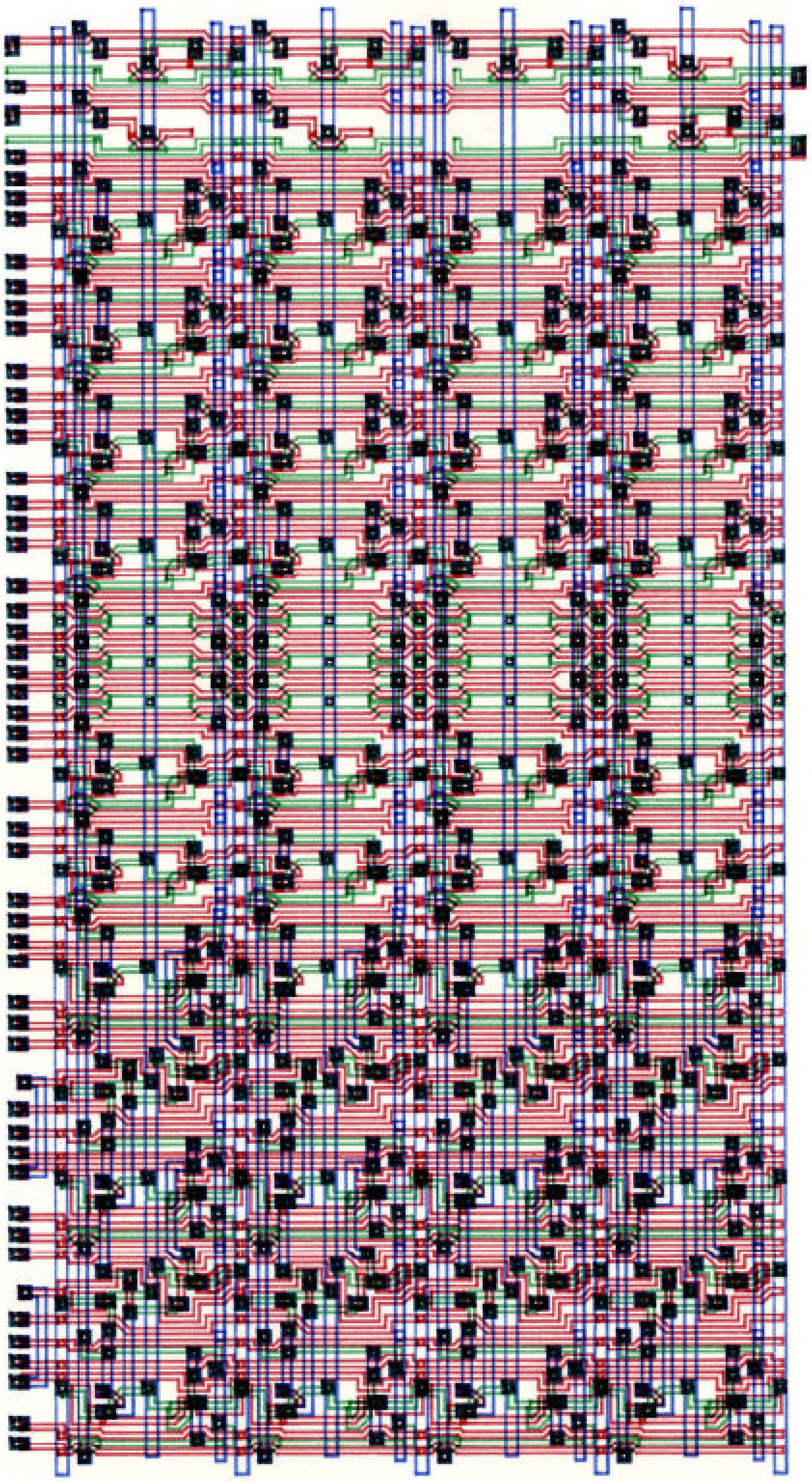


Figure 5b: Local Stack Controller Layout

Figure 5 shows the components of a typical local stack controller. Note that only four bits of the 16 bit datapath are shown. The block diagram is drawn at the same scale as the layout. These conventions are followed throughout this thesis. In figure 5, the stack message CAM contains the byte codes which are executed by the local stack. The first byte of each message is loaded into the CAM to see if it should be acted on. The accepted and completed registers contain a code indicating how many instructions have been accepted and completed, respectively. This information is used by the execution section of the chip. The stack *valid* register keeps track of the number of *valid* words on the *local* stack. The constants include 1, 0, -1, the local stack depth, the STACK UNDERFLOW message header, and the POP STACK 0 message header. Finally, the local stack controller includes a subtractor and the local stack itself. Table 4 lists the messages recognized by the local stack controller and the actions performed.

Message	Action
Accept Instr	Increment accepted register
Pop Stack N	Pop *n* words off the local stack, push following data, and increment completed register
Restore Stack	Invalidate the local stack and push following data
Reset	Invalidate the local stack, initialize accepted and completed

Table 4: Local Stack Unit Messages

The datapath operations required to execute the above messages are very simple. For example, the ACCEPT INSTR message causes the value in the accepted register to be loaded into the subtractor, incremented, and restored to the accepted register. The POP STACK N message is not much more complicated. Given this, it would be wasteful to use an entire microcode sequencer to perform these operations. An on-chip, clocked PLA would be sufficient. However, this is not implemented in the present version of Bristle Blocks. The present instruction decoder is nothing more than the AND plane of a PLA, with clocking and latching provided in the datapath buffers. Designing a more general instruction decoder is an important area for further work on Bristle Blocks.

6.3 Instruction Scanner

Each processor except the instruction unit contains an instruction scanner. Its function is to scan the stack bus message stream searching for instructions to execute. When such an instruction is recognized, the instruction scanner saves the operand, if any, and arbitrates for control of the stack bus. While waiting to get control of the stack bus, the instruction scanner scans the message stream looking for an ACCEPT INSTR message. If one is found, another processor has accepted that instruction, so the scanner goes back to searching the message stream for an instruction to execute. If the scanner is given control of the bus before an ACCEPT INSTR message has been sent, then that processor accepts the instruction. This is done by saving the present value of the accepted register from the local stack unit and then sending an ACCEPT INSTR message.

In the present version of the Pascal system, no instruction is recognized by more than one processor, so scanning for ACCEPT INSTR is not strictly necessary. To remain compatible with future versions for which this may not be true, and because it does not cost any execution speed, it would be better to scan for ACCEPT INSTR anyway. Some instructions, such as those to the stack manager and memory manager, must be recognized within only one processor. In these cases the instruction scanner should declare a runtime error if another ACCEPT INSTR message is found.

After an instruction has been accepted, the processor does whatever setup is necessary to execute the instruction. However, the stack may not be read until all prior instructions have completed execution. This condition is true when the completed register in the local stack unit is equal to the saved value of the accepted register. At this time the processor is free to read values from the local stack, compute a result, and send a POP STACK N message over the stack bus. This increments the completed register in each local stack controller, allowing the next instruction to be executed.

Sometimes the stack depth may become too great, causing the lowermost words to be pushed off the bottom of a local stack. This condition is detected when an processor reads a value from its local stack and finds that it is marked invalid. When this happens, the processor issues a STACK UNDERFLOW message and waits in a loop, periodically re-issuing the message. This message is received by the stack manager, which responds by issuing a RESTORE STACK message. After the processor detects the restore stack message, execution continues.

Figure 6 shows the components of a typical instruction scanner. The instruction CAM is divided into two sections. The first section is used to screen the first byte of messages to see if they are recognized instructions. The second section is used to screen the second byte of instruction messages which began with an escape bytecode. The auxiliary CAM recognizes the escape bytecode and also recognizes the TAKE INSTR message header. The bytecodes unit is a constant source, containing the bytecodes for the ACCEPT INSTR, STACK UNDERFLOW, and POP STACK N messages. It also contains the constant zero. Finally, accept compare is a CAM which recognizes when the saved value of the accepted register matches the present value of the completed register.

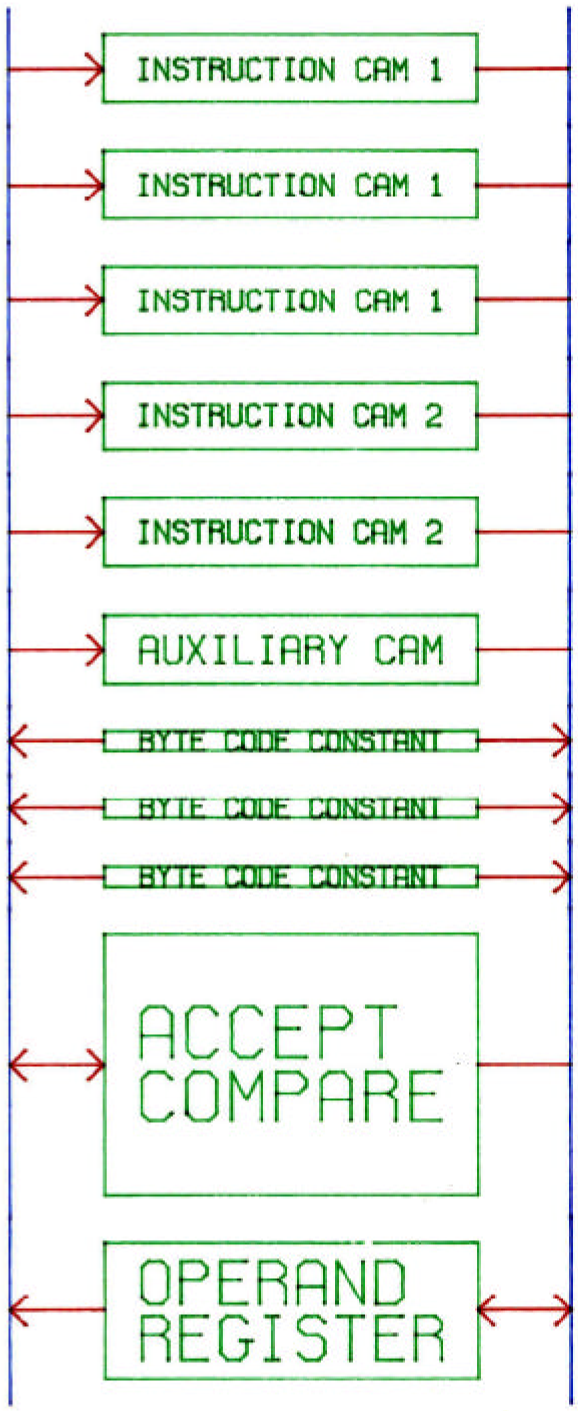


Figure 6a: Instruction Scanner Block Diagram

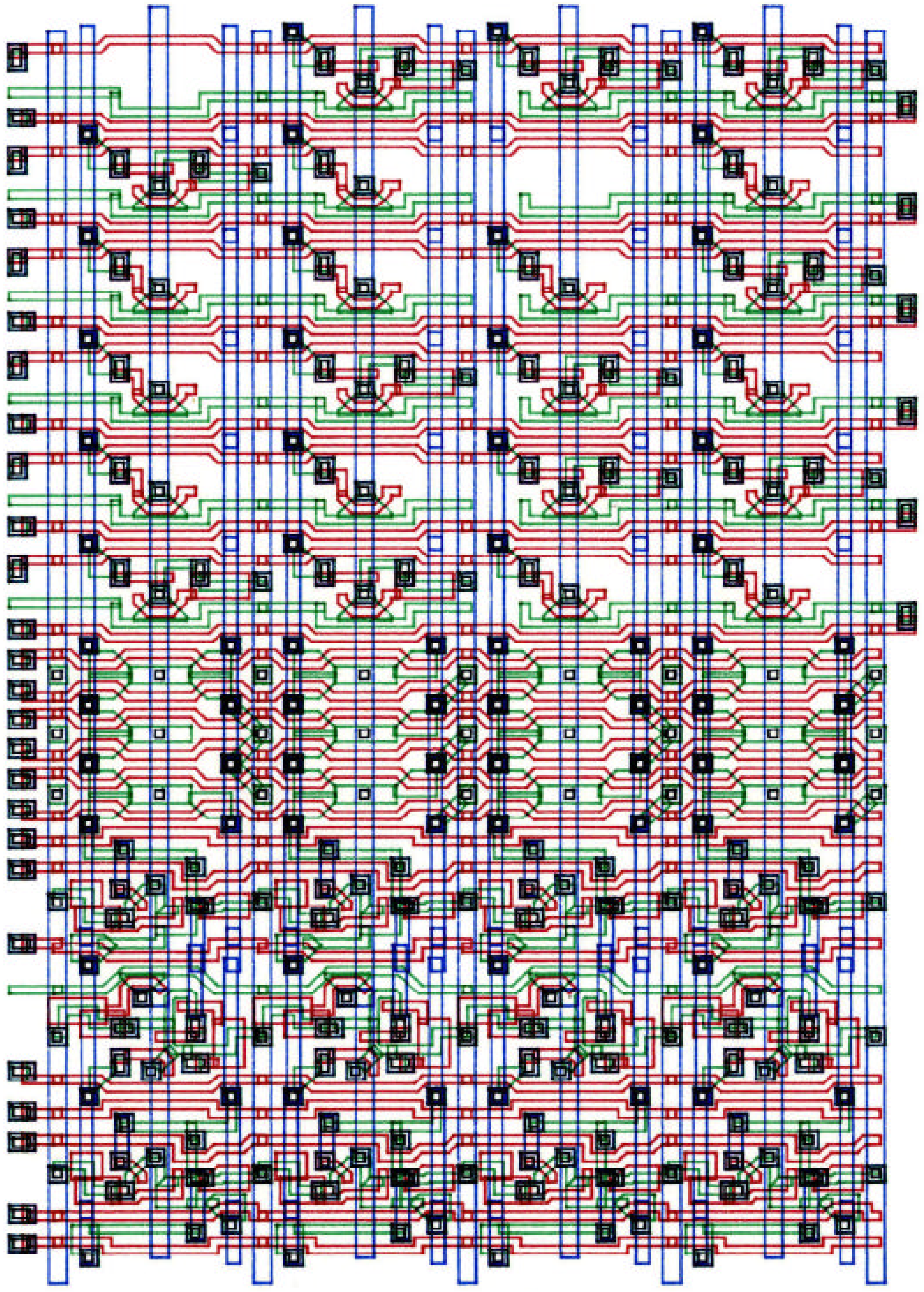


Figure 6b: Instruction Scanner Layout

The datapath operations performed by the instruction scanner are even simpler than those performed by the local stack controller. The instruction scanner could easily have its own on-chip microcode sequencer in a future version of Bristle Blocks. This is not necessary, though, because instruction scanning and instruction execution are mutually exclusive activities. Therefore, the instruction scanner could use the same sequencer as the execution section of the processor. This sequencer could be on-chip or off-chip, depending on the complexity of the processor.

7. Arithmetic and Stack Processors

This section describes three of the processors in the Pascal system: the integer processor, the floating point processor, and the stack manager.

7.1 Integer Processor

The integer processor executes the single word and double word integer arithmetic and compare instructions. These instructions are listed in table 5 below. Their frequencies total to 7.13%. Figure 7 shows the internal structure of the integer processor. The stack bus interface, local stack unit, and instruction accept unit are described in the previous section. No integer instruction requires more than four words from the stack and most require two or less. Therefore, the integer processor's local stack need only be four words deep.

Opcode, Operand		Instr. Group	Code	Static Freq.	Percent	Instruction Description
add		integer		1469	0.79 %	Integer add
sub		integer		878	0.47 %	Integer subtract
mul		integer		565	0.30 %	Integer multiply
div		integer		383	0.21 %	Integer divide
mod		integer		343	0.18 %	Modulo (remainder)
neg		integer		108	0.06 %	Negate (two's complement)
shl		integer		139	0.07 %	Shift left tos-1 by tos bits
shr		integer		0	0.00 %	Shift right tos-1 by tos bits
rol		integer		0	0.00 %	Rotate left tos-1 by tos bits
ror		integer		0	0.00 %	Rotate right tos-1 by tos bits
inc		integer		797	0.43 %	Increment top of stack by 1
dec		integer		478	0.26 %	Decrement top of stack by 1
exg		integer		0	0.00 %	Exchange top two words
adi	b	integer		960	0.51 %	Add the constant b to top of stack; do not check overflow
ads		integer		0	0.00 %	Same as add, but do not check for overflow
dad		double		0	0.00 %	Double add
dsb		double		0	0.00 %	Double subtract
dmu		double		0	0.00 %	Double multiply
ddv		double		0	0.00 %	Double divide
dmd		double		0	0.00 %	Double modulo
cid		convert		0	0.00 %	Convert integer to double
cdi		convert		0	0.00 %	Convert double to integer
and	2	boolean	m	1035	0.55 %	Boolean and on two groups of 2 bytes
ior	2	boolean	m	726	0.39 %	Boolean inclusive or on two groups of 2 bytes
not		boolean		0	0.00 %	Convert top of stack from true to false or vice versa
cmi		compare		1911	1.02 %	Compare two integers. Push -1,0,1 for <, = ,>
cmd		compare		0	0.00 %	Compare two double integers
tlt		compare		292	0.16 %	True if less (based on previous compare)
tle		compare		172	0.09 %	True if less or equal
teq		compare		1184	0.63 %	True if equal
tne		compare		591	0.32 %	True if not equal
tge		compare		134	0.07 %	True if greater or equal
tgt		compare		290	0.16 %	True if greater
dup	2	misc	m	845	0.45 %	Duplicate top 2 words on stack

Table 5: Integer Processor Instructions

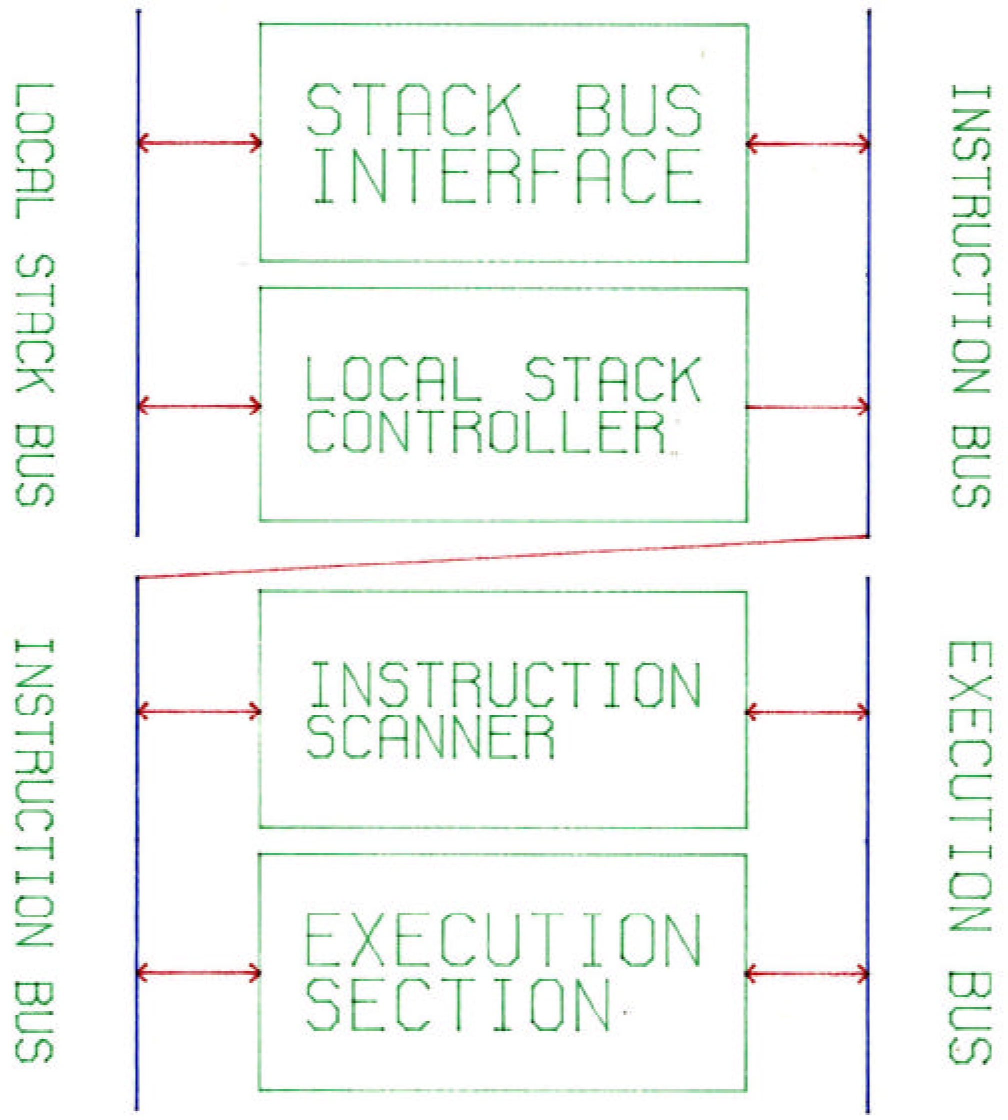


Figure 7: Integer Processor Block Diagram

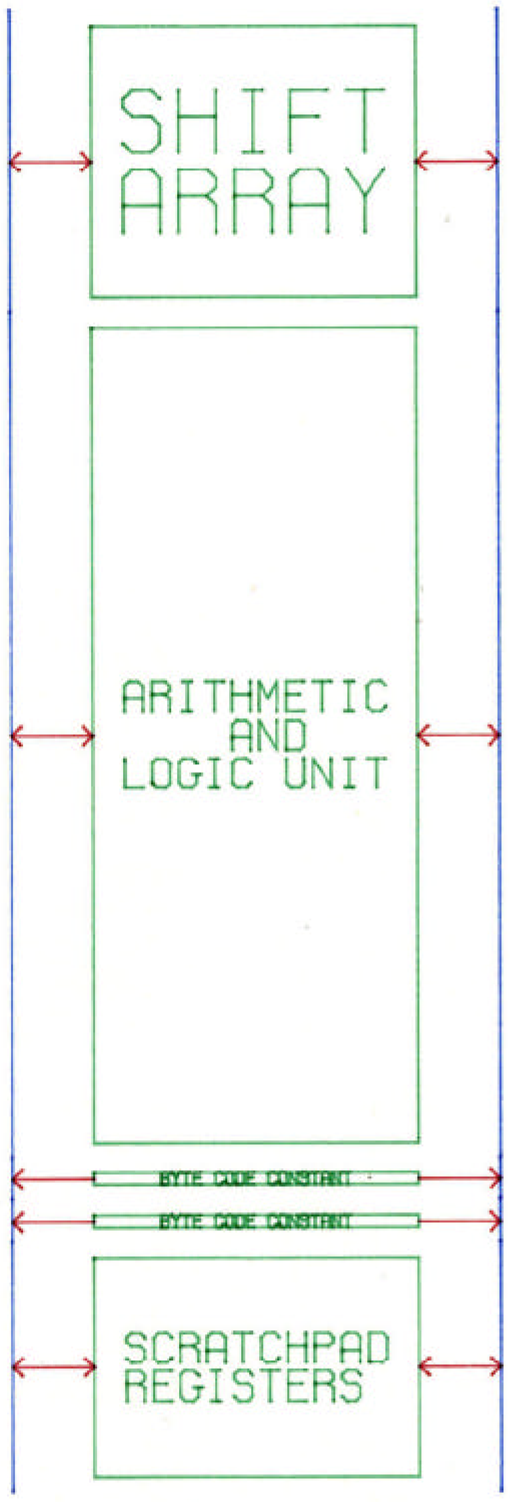


Figure 8a: Integer Processor Execution Section Block Diagram

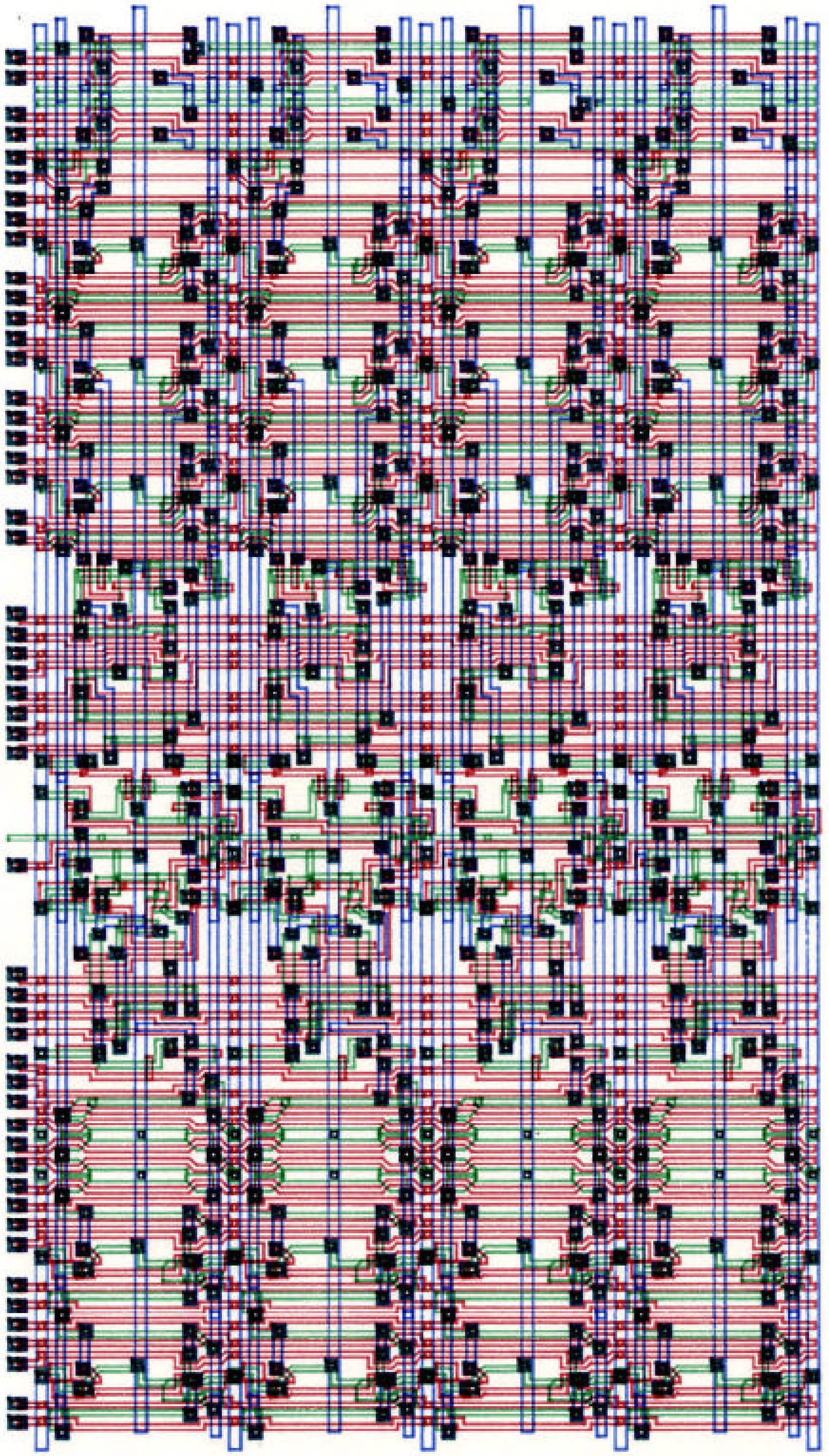


Figure 8b: Integer Processor Execution Section Layout

Figure 8 shows the internal structure of the execution section. To execute its instructions, the integer processor needs only a shift array, an ALU, some numeric constants (0, 1, -1, and $-\infty$, that is, 100...), and some scratch registers. The datapath operations required to execute these instructions are simple. Therefore, this execution unit could use an on-chip microcode sequencer.

7.2 Floating Point Processor

The floating point processor executes the floating point and conversion instructions listed in table 6 below. Their frequencies total to 1.77%. A detailed design of the floating point processor is beyond the scope of this thesis. It should be noted that the integer and floating point processors could be merged into a single chip in a future version of the Pascal system. The low frequencies of the instructions executed by both chips would lead one to do this. A specialized floating point chip could then be designed for use in applications that make significant use of floating point operations.

Opcode, Operand	Instr. Group	Code	Static Freq.	Percent	Instruction Description
fad	real		355	0.19 %	Floating add
fsb	real		301	0.16 %	Floating subtract
fmu	real		482	0.26 %	Floating multiply
fdv	real		289	0.15 %	Floating divide
cif	convert		1227	0.66 %	Convert integer to floating
cfi	convert		303	0.16 %	Convert floating to integer
cdf	convert		0	0.00 %	Convert double to floating
cfd	convert		0	0.00 %	Convert floating to double
cmf	compare		338	0.18 %	Compare two reals

Table 6: Floating Point Processor Instructions

7.3 Stack Manager

The stack manager has a dual purpose. It interfaces the local stacks with the stack stored in main memory, saving data pushed out the bottom of its local stack and restoring it when the stack depth decreases again. It also executes instructions which manipulate the stack or multiword operands on the stack. The instructions it executes are listed in table 7 below. Their frequencies total to 13.56%.

Opcode, Operand		Instr. Group	Code	Static Freq.	Percent	Instruction Description
and	x	boolean		55	0.03 %	Boolean and on two groups of x bytes
ans		boolean		0	0.00 %	Boolean and; first pop number of bytes from stack
ior	x	boolean		325	0.17 %	Boolean inclusive or on two groups of x bytes
ios		boolean		0	0.00 %	Boolean inclusive or; first pop number of bytes from stack
xor	x	boolean		0	0.00 %	Boolean exclusive or on two groups of x bytes
xos		boolean		0	0.00 %	Boolean exclusive or; first pop number of bytes from stack
com	x	boolean		48	0.03 %	Complement (one's complement of top x bytes)
cos		boolean		0	0.00 %	Complement; first pop number of bytes from stack
inn	x	set		363	0.19 %	Bit test on x byte set (bit number on top of stack)
ins		set		0	0.00 %	Bit test; first pop set size, then bit number
set	x	set		119	0.06 %	Create singleton x word set with bit number tos on
ses		set		0	0.00 %	Create singleton set; first pop set size, then bit number
cmu	x	compare		35	0.02 %	Compare two blocks of x bytes each
cms		compare		0	0.00 %	Compare two blocks of bytes; pop byte count
cal	n	call	!	19147	10.26 %	Call procedure with descriptor n
cas		call	!	4	0.00 %	Call indirect (first pop procedure number from stack)
ret	n	call	!	3517	1.88 %	Return (function result consists of top x bytes)
res		call	!	0	0.00 %	Return; first pop number of bytes to return
beg	x	misc		1613	0.86 %	Begin procedure (reserve x bytes for locals)
bes		misc		0	0.00 %	Begin procedure; pop number of bytes to reserve for locals
dup	x	misc		79	0.04 %	Duplicate top x words on stack
dus		misc		0	0.00 %	Duplicate; first pop number of words to duplicate
lor	3	misc	m	0	0.00 %	Load stack pointer register
str	3	misc	m	0	0.00 %	Store stack pointer register

Table 7: Stack Manager Instructions

Figure 9 shows the internal structure of the stack manager. Its local stack controller is somewhat special, because it also needs to keep track of when data is going to be lost out the bottom of the local stack. When this happens, the local stack is written out to main memory and the stack pointer is updated. The local stack is also written out to memory on receipt of a load stack pointer instruction (lor 3). This instruction is sent by the instruction unit when the memory manager needs to access data relative to the stack pointer, which is necessary in order to execute instructions such as store indirect (sti y) and store array element (sar x). Instructions for which this is done are marked with "*" in their code fields.

Figure 10 shows the internal structure of the stack manager execution section. All that is needed is an ALU, some scratch registers, the stack pointer register, and some constants. The constants consist of the numeric constants (0,1,-1,and $-\infty$) and the message headers necessary to communicate with the memory bus.

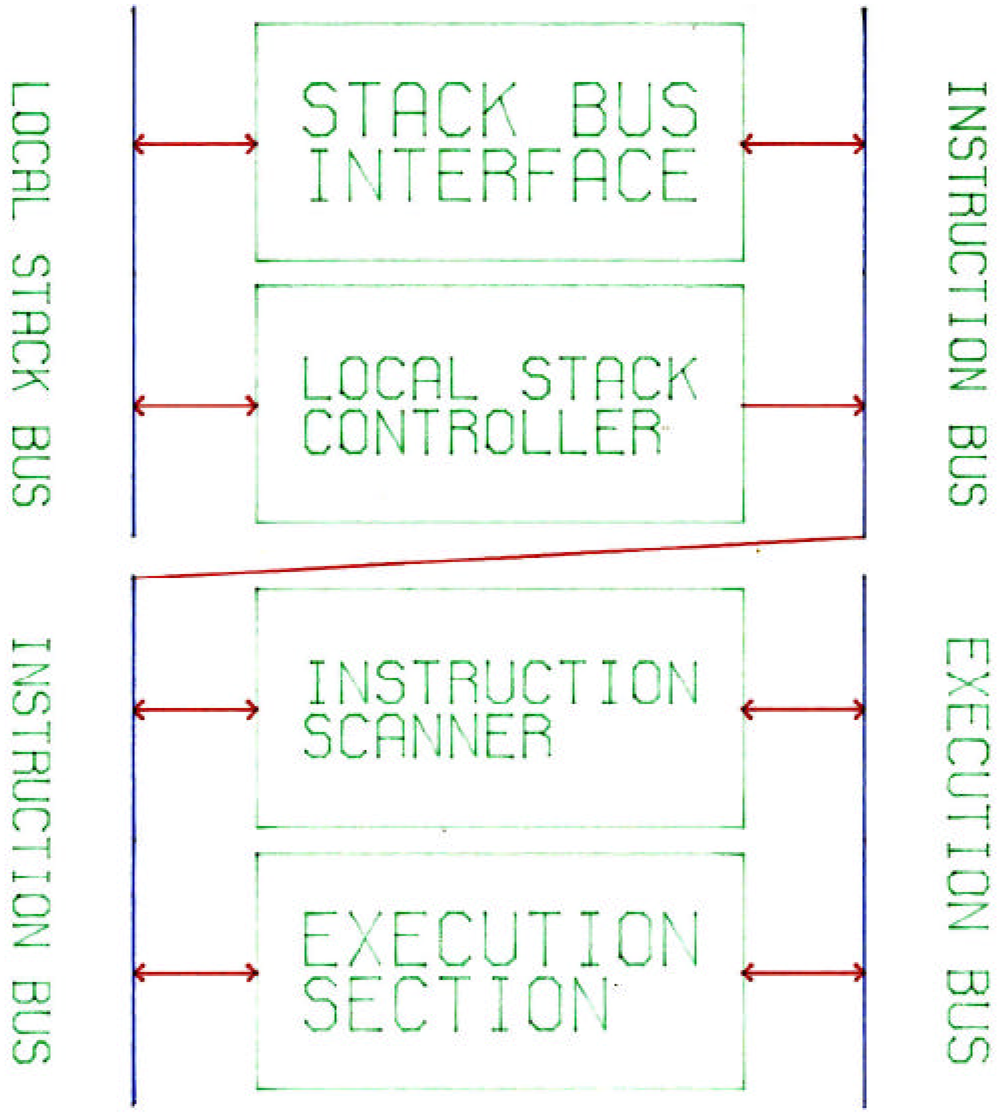


Figure 9: Stack Manager Block Diagram

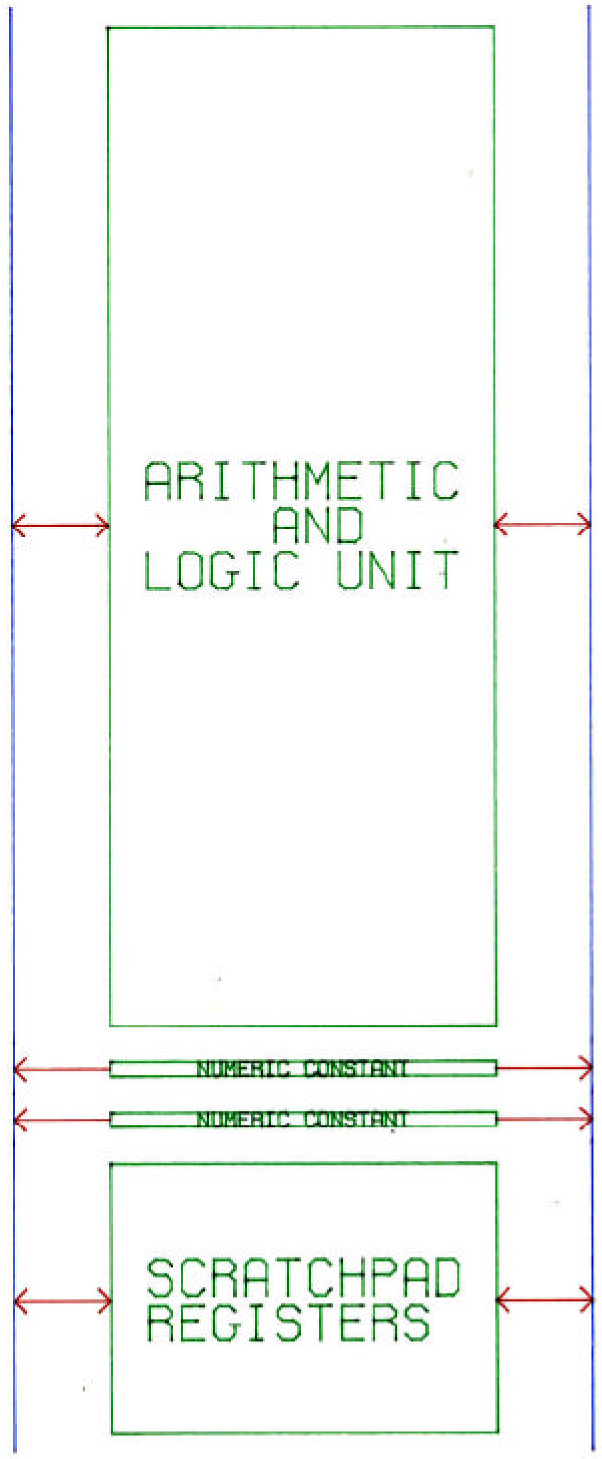


Figure 10a: Stack Manager Execution Section Block Diagram

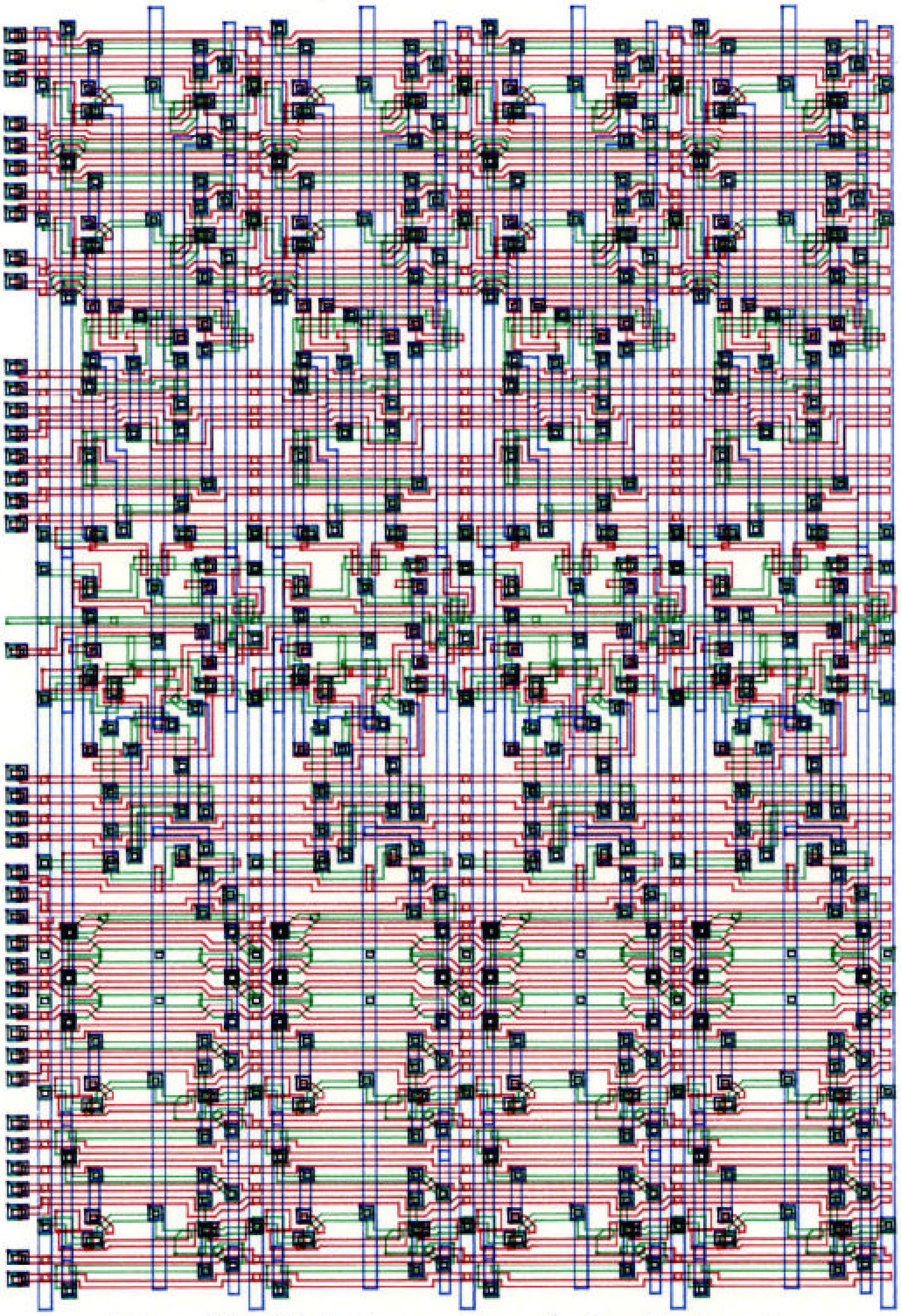


Figure 10b: Stack Manager Execution Section Layout

8. Memory Manager

The memory manager executes all instructions which access variables in main memory. These instructions are listed in table 8. Their frequencies total to 53.33%. Figure 11 shows the internal structure of the memory manager; the various sections are described below.

Opcode.		Instr.		Static		
Operand		Group	Code	Freq.	Percent	Instruction Description
lol	x	load		17140	9.18 %	Load local word x
loe	x	load		9490	5.08 %	Load external word x
lop	x	load		676	0.36 %	Load parameter (address is at xth local)
lof	x	load		3472	1.86 %	Load offsetted (top of stack + x yields address)
lal	x	load		1805	0.97 %	Load address of local
lae	x	load		16242	8.70 %	Load address of external
lex	n	load		1885	1.01 %	Load lexical (address of lb, n static levels back)
loi	y	load		1561	0.84 %	Load indirect y bytes (address is popped from the stack)
lib		load	m	0	0.00 %	Load indirect 1 byte (loaded into 1 word on stack)
los		load		0	0.00 %	Load indirect (first pop byte count; must be 1 or even)
ldl	x	load		1576	0.84 %	Load double local (two consecutive locals are stacked)
lde	x	load		1196	0.64 %	Load double external
ldf	x	load		73	0.04 %	Load double offsetted (top of stack + x yields address)
stl	x	store		6587	3.53 %	Store local
ste	x	store		3588	1.92 %	Store external
stp	x	store		731	0.39 %	Store parameter
stf	x	store		1788	0.96 %	Store offsetted
sti	y	store	*	746	0.40 %	Store indirect y bytes (pop address, then data)
sib		store	m	0	0.00 %	Store indirect 1 byte (taken from 1 word on stack)
sts		store	*	0	0.00 %	Store indirect (pop byte count, then address, then data)
sdl	x	store		377	0.20 %	Store double local
sde	x	store		390	0.21 %	Store double external
sdf	x	store		60	0.03 %	Store double offsetted
lar	x	array		2111	1.13 %	Load array element with descriptor x, pop index, array addr
las		array		0	0.00 %	Load array element; first pop pointer to descriptor
sar	x	array	*	1245	0.67 %	Store array element; pop index, array address, array element
sas		array	*	0	0.00 %	Store array element; first pop pointer to descriptor
aar	x	array		1753	0.94 %	Stack address of array element; pop index, array address
aas		array		0	0.00 %	Stack array address; first pop pointer to descriptor
mrk	n	call		19147	10.26 %	Mark stack (n = 1 + change in static depth of nesting)
mrx	n	call	m	0	0.00 %	Mark stack; set alternate context
mrs		call		4	0.00 %	Mark stack; first pop static link from stack
mxs		call	m	0	0.00 %	Mark stack; pop static link and set alternate context
inl	x	inc/dec		1403	0.75 %	Increment local
ine	x	inc/dec		1285	0.69 %	Increment external
del	x	inc/dec		126	0.07 %	Decrement local
dee	x	inc/dec		71	0.04 %	Decrement external
zrl	x	inc/dec		784	0.42 %	Zero local
zre	x	inc/dec		760	0.41 %	Zero external
rck	x	misc		987	0.53 %	Range check (trap if top of stack out of range)
blm	x	misc		482	0.26 %	Block move x bytes; pop source address, then destination
bls		misc		0	0.00 %	Block move; first pop x, then addresses
lor	1	misc	m	0	0.00 %	Load procedure descriptors register
lor	2	misc	m	0	0.00 %	Load local base register
lor	4	misc	m	0	0.00 %	Load heap pointer register
str	2	misc	m	0	0.00 %	Store local base register
str	4	misc	m	0	0.00 %	Store heap pointer register

Table 8: Memory Manager Instructions

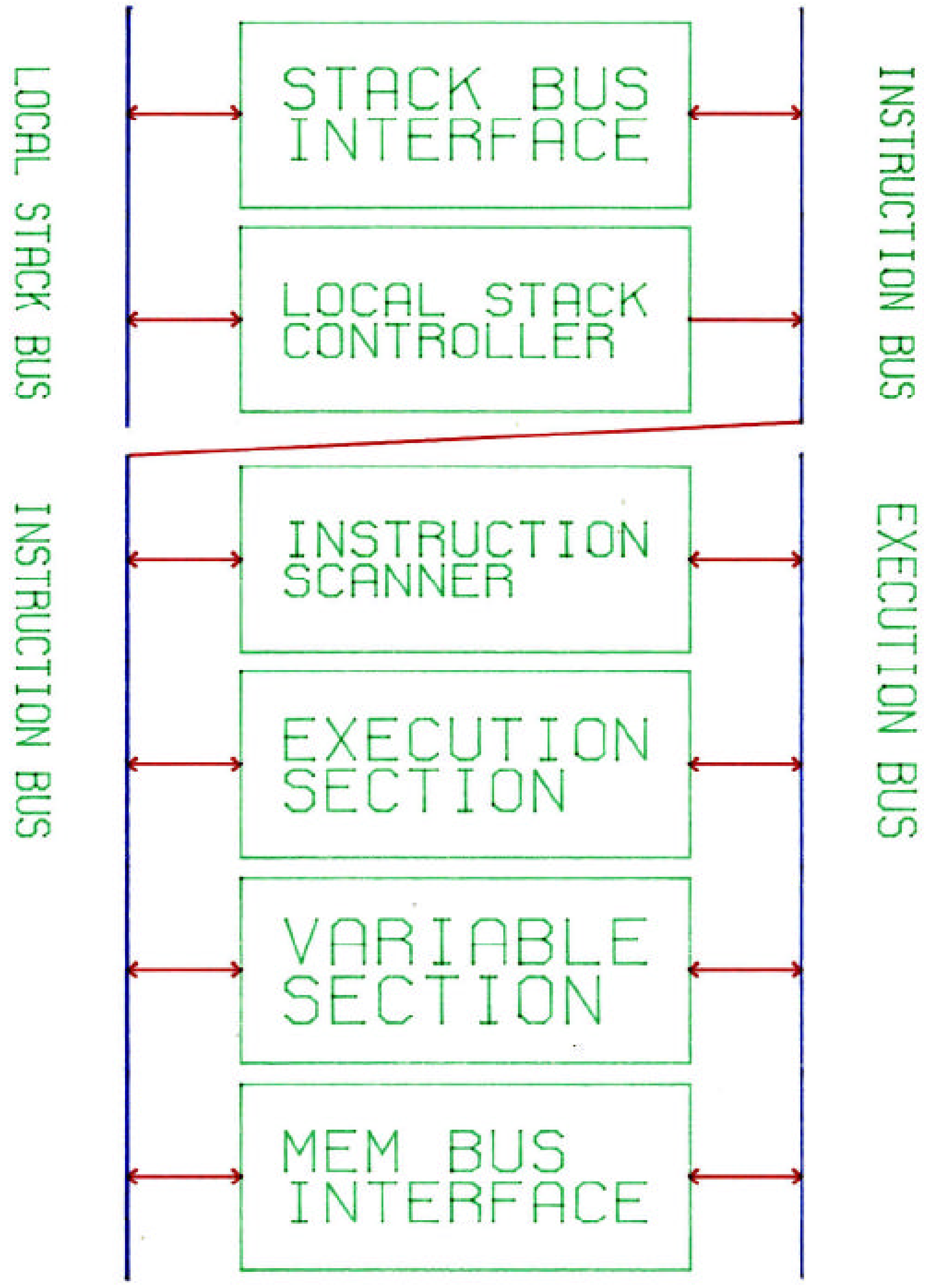


Figure 11: Memory Manager Block Diagram

8.1 Variables Section

The variables section exists in recognition of the fact that most memory accesses in a block structured language are made to the first few local and global (or external) variables. The variables section consists of 16 memory registers. Eight of them are reserved for the first eight global variables, and the other eight are reserved for the first eight local variables. All reads and writes in these ranges are done exclusively on-chip. In this way, a large fraction of variable accesses do not need to use main memory cycles. Using the frequency data in section 5 of appendix B, the first eight local variables account for 14,799 out of 17,140 static occurences of the load local instruction (lol x). This means that 86% of local variable accesses do not need to access main memory. The data for the load external instruction (loe x) is not as good: only 1998 out of 9490, or 21% of the instructions load from the first eight global variables. The reason why this figure is so much poorer than for the load local instruction is that there are many more global (external) variables than local variables. Since the variables are defined in an arbitrary order, it seems reasonable to have the assembler order the globals such that those variables which are accessed often occur first. For Tannenbaum's frequency data this would only increase the incidence to 30%, so such a scheme is probably not worth implementing.

One disadvantage of the variable registers as described above is that they must be saved on each procedure call and restored on each procedure return. This is not as much of a disadvantage as it would be for less concurrent architectures, because the saves and restores are not likely to be on the critical path. If this turns out to be a problem, a more complex scheme could be used which would eliminate much of the saving and restoring. For example, a write-through policy would remove the need to save the registers on procedure calls. Including valid bits for the variable registers would eliminate the need to restore them on a procedure return; they would simply be marked invalid. The memory manager would be free to fill them in during its spare time.

8.2 Execution Section and Instruction Scanner

The execution section of the memory manager is detailed in figure 12. It contains the local and external base registers, the heap pointer, the memory limit register, some constants, and a subtractor. It also contains an instruction CAM, which the previously described processors do not have. Instead of getting instructions directly from the instruction scanner, the execution section reads them from a buffer contained in the instruction scanner. The execution section's instruction CAM is for interpreting these buffered instructions.

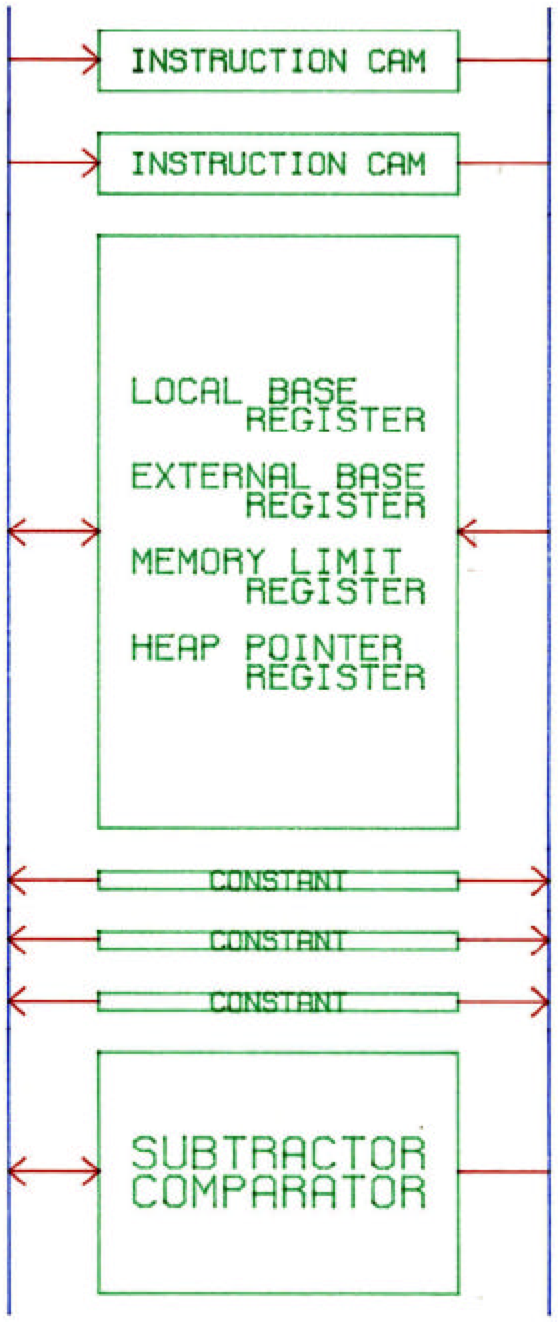


Figure 12a: Memory Manager Execution Section Block Diagram

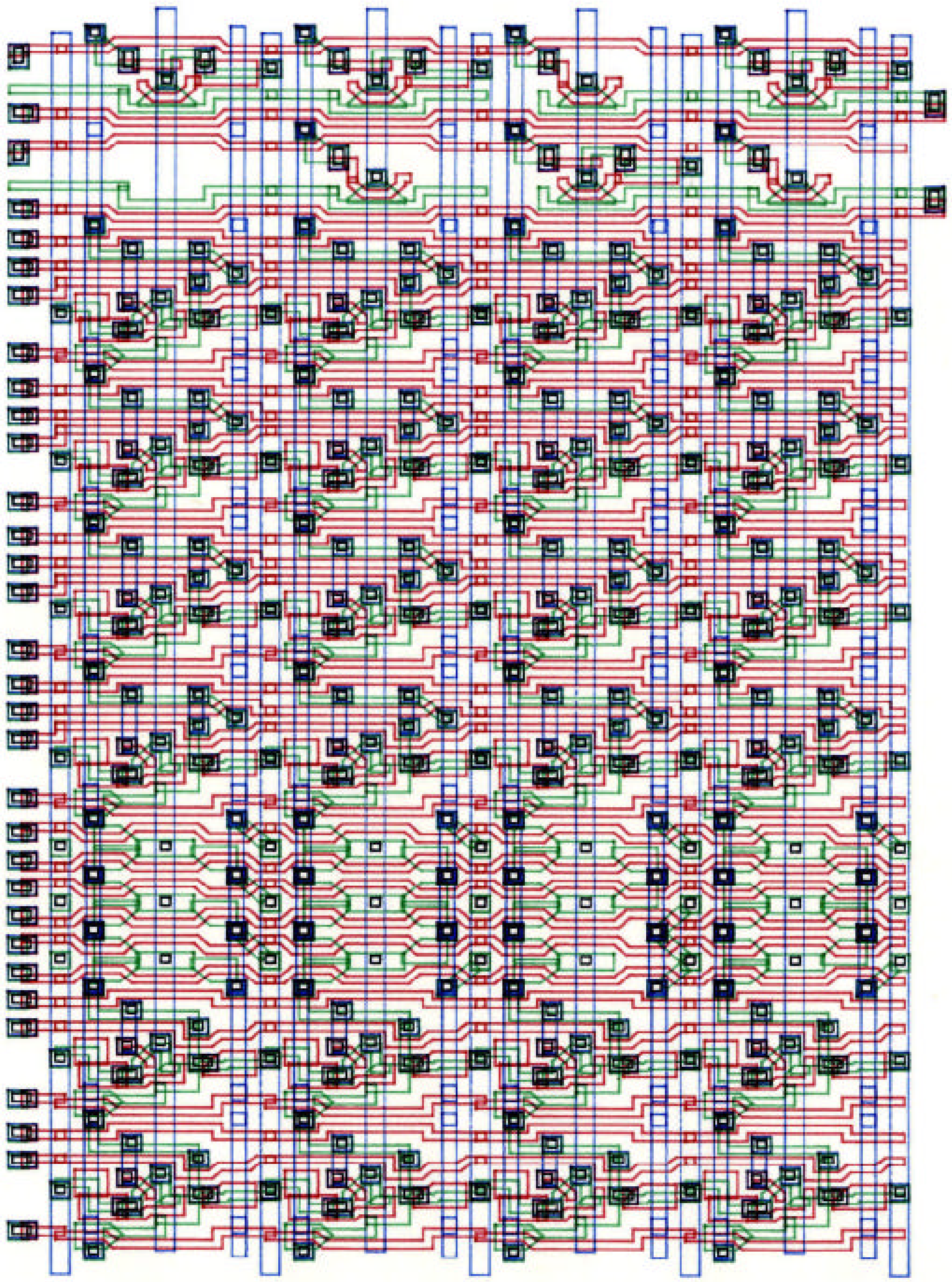


Figure 12b: Memory Manager Execution Section Layout

The memory manager's instruction scanner buffers the instructions it accepts because the memory manager cannot be limited to accepting only one instruction at a time. There are two reasons why it cannot. First, too many instructions are executed in the memory manager. These instructions account for 53% of the static frequency. The system would lose most of its concurrency if only one could be accepted at a time. Second, many memory manager instructions can be partially executed before the prior instructions have completed. For example, a load array element instruction (lar x) can read its descriptor and calculate the array element address as soon as the instruction is received. The array element may be read as soon as all prior incomplete instructions which could write to that address have finished execution. This removes main memory accesses from the critical path, which is an important result.

The memory manager is also different from the previously described processors in that its instruction scanner has a sequencer of its own. This is necessary because the instruction scanner continually scans for instructions to accept rather than stopping after each instruction. When it finds an instruction it can accept, it loads it into the instruction buffer. Prior to loading an instruction and its operand into the instruction buffer, the scanner loads the current value of the accepted register from the local stack controller. This is necessary so that the execution section knows when to complete execution of the instruction.

The instruction buffer allows words to be inserted and deleted at any point, rather than just at the ends as in a stack or a fifo. This added functionality gives the execution sequencer quite a bit of freedom in how it can execute its instructions. Instructions can be popped out of the middle of the buffer and replaced by other instructions, if the execution algorithm calls for it. It is interesting to note that this added functionality complicates the decoder section of the Bristle Blocks chip, but does not affect the design of the datapath element.

8.3 Execution Algorithm

The choice of the specific execution algorithm used by the execution section of the memory manager is a tradeoff between complexity and concurrency: the more complex the algorithm, the more it is able to take advantage of potential concurrency by executing parts of instructions in advance. The algorithm described below clarifies some of the tradeoffs to be made.

To start with, the first word in the instruction buffer, which is a value saved from the accepted register, is compared with the completed register in the local stack controller. If they match, the following instruction can now be executed. If they don't match, it may be that the instruction can be partially executed. For example, a store parameter instruction (stp x) could have its store address fetched. The store parameter instruction would then be replaced in the instruction buffer with a store instruction for that address. The idea is to do as much calculation as possible ahead of time, thus moving it off the critical path.

A simple execution algorithm could stop here. However, under certain conditions it is safe to partially execute instructions which occur farther back in the buffer. For example, suppose a load offsetted instruction (lof x) is followed by a load external instruction (loe x). Nothing can be done with the load offsetted instruction until all prior instructions have completed. While waiting for that to happen, the processor could read in the external value and change the load external instruction into a load constant instruction, which will execute much faster.

Depending on the amount of pre-execution desired, the execution algorithm can be made arbitrarily complex. It would be possible to have status flags which are set by certain instructions to control which later instructions are allowed to partially execute. For example, a load local instruction could not be pre-executed if an earlier instruction had set a store local flag. The store local base instruction (str 2) could set a flag which stops all local loads and stores from executing. Of course, the processor should go back at each step and check whether the first word in the buffer matches the completed register. If it does, the first instruction in the buffer should be executed immediately, because we are now on the critical path. After completing execution of an instruction, the flags would be reset and the algorithm would start over from the beginning.

In choosing the execution algorithm, it is important to keep instruction frequencies and system performance firmly in mind. For example, it does no good to have a highly complex, *n* levels deep pre-execute algorithm if there are never more than two instructions in the buffer. It does little good to include special cases to handle seldom executed instructions. Especially in an early version, the simplest algorithm that satisfies the recognized requirements and limitations of the system is best. For this reason, the memory manager execution algorithm for the first version of the Pascal system will only attempt to pre-execute the first instruction in the buffer.

9. Instruction Unit

The instruction unit fetches instructions from main memory, decodes them, and issues them across the stack bus to the various processors. Some instructions are executed by the instruction unit itself. These are not issued across the the stack bus, although an INSTR ACCEPT message is issued. The instructions executed by the instruction unit are listed in table 9 below. Their frequencies total 4.22%. The instructions marked with an "x" in the code field do not even have an INSTR ACCEPT message issued. They are executed completely within the instruction unit.

Opcode,	Operand	Instr. Group	Code	Static Freq.	Percent	Instruction Description
loc	m	load		27604	14.79 %	Load constant (i.e. push it onto the stack)
lnc	n	load	m	1136	0.61 %	Load negative constant
brf	n	branch	x	3067	1.64 %	Branch foreward unconditionally n bytes
brb	n	branch	x	2688	1.44 %	Branch backward unconditionally n bytes
blt	n	branch		1016	0.54 %	Forward branch less (pop two words, branch if top > second)
ble	n	branch		305	0.16 %	Forward branch less or equal
beq	n	branch		1060	0.57 %	Forward branch equal
bne	n	branch		1681	0.90 %	Forward branch not equal
bge	n	branch		293	0.16 %	Forward branch greater or equal
bgt	n	branch		598	0.32 %	Forward branch greater
zlt	n	branch		47	0.03 %	Forward branch less than zero (pop 1 word, branch negative)
zle	n	branch		217	0.12 %	Forward branch less or equal to zero
zeq	n	branch		2094	1.12 %	Forward branch equal zero
zne	n	branch		1139	0.61 %	Forward branch not zero
zge	n	branch		155	0.08 %	Forward branch greater or equal to zero
zgt	n	branch		31	0.02 %	Forward branch greater than zero
nop		misc	x	0	0.00 %	No operation
lin	n	misc		1763	0.94 %	Line number (set external 0 to n)
cse	x	misc		297	0.16 %	Case jump; x is external offset of jump table
lor	n	misc		0	0.00 %	Load EM-1 machine register onto stack
str	n	misc		0	0.00 %	Store EM-1 machine register from stack
str	1	misc	m	0	0.00 %	Store procedure descriptor register
hlt		misc		0	0.00 %	Halt the machine
mon		monitor		5	0.00 %	Monitor call
stu		monitor		0	0.00 %	Start user job

Table 9: Instruction Unit Instructions

9.1 Internal Structure

Figure 13 shows the internal structure of the instruction unit. On the left side, the chip interfaces to the memory bus and on the right it interfaces to the stack bus. The local stack controller has been described previously; the prefetch, decode, and execution sections are described below.

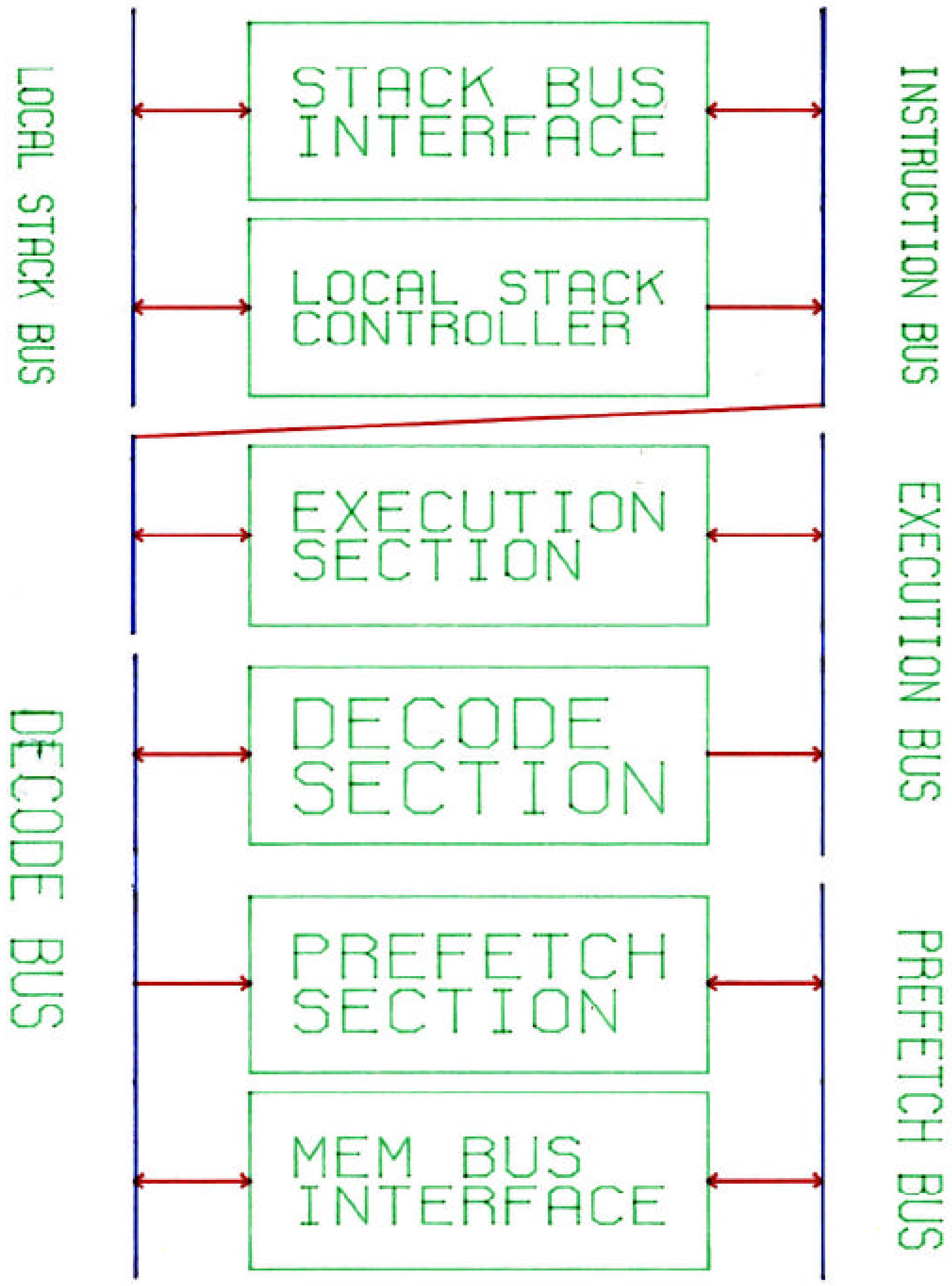


Figure 13: Instruction Unit Block Diagram

The prefetch section holds instructions that have been fetched from main memory before they are decoded. The components of the prefetch section are two fifo buffers, each of which has a byte manipulator on its output end. The byte manipulator is similar in function to the byte manipulator contained in the stack bus interface, and has the same purpose. The two fifo buffers are filled in alternation: one is filled while instructions are being read from the other. The buffer length should be chosen such that the time required to fill one of the buffers is less than the time required to issue all the instructions from one of the buffers. In the present system, the buffer length is chosen to be four. A better value could be chosen in a future system by measuring the performance of the system with this buffer size.

The decode section reads instructions from the prefetch buffers and decodes them to determine their length and whether they are executed within the instruction unit. Figure 14 shows the components of the decode section. The instruction length CAMs are used to find the length in bytes of the various instructions. There are two of them in order to handle normal and escaped instructions. The instruction execute CAMs are used to find instructions to be executed within the instruction unit. The purpose of the procedure descriptor, procedure base and program status registers is given in the EM-1 machine definition. The program counter register is incremented each time a byte is read from one of the prefetch buffers. The function of the return address register is described below in the section on the procedure call and return mechanism. Finally, the decode section contains a subtractor and a constant source.

The execute section saves constants to be loaded onto the stack and issues them across the stack bus. It contains a subtractor/comparator, a constant source, and the the load constant buffer. The constant source contains the message headers necessary to send messages on the stack bus. The load constant buffer is a fifo buffer used in executing the load constant instructions. When a load constant (loc m) or load negative constant (lnc n) instruction is decoded, the value of the accepted register in the local stack controller is loaded into the load constant buffer, followed by the constant to be loaded. Then an ACCEPT INSTR message is sent over the stack bus. The execution section compares the first word in the fifo to the completed register in the local stack controller. When they are equal, a POP STACK 0 message is sent which pushes the next word in the fifo onto the stack. In this way constants can be loaded onto the stack without stopping the instruction prefetch sequence. This is important, since the static frequency of the load constant and load negative constant instructions total more than 15%.

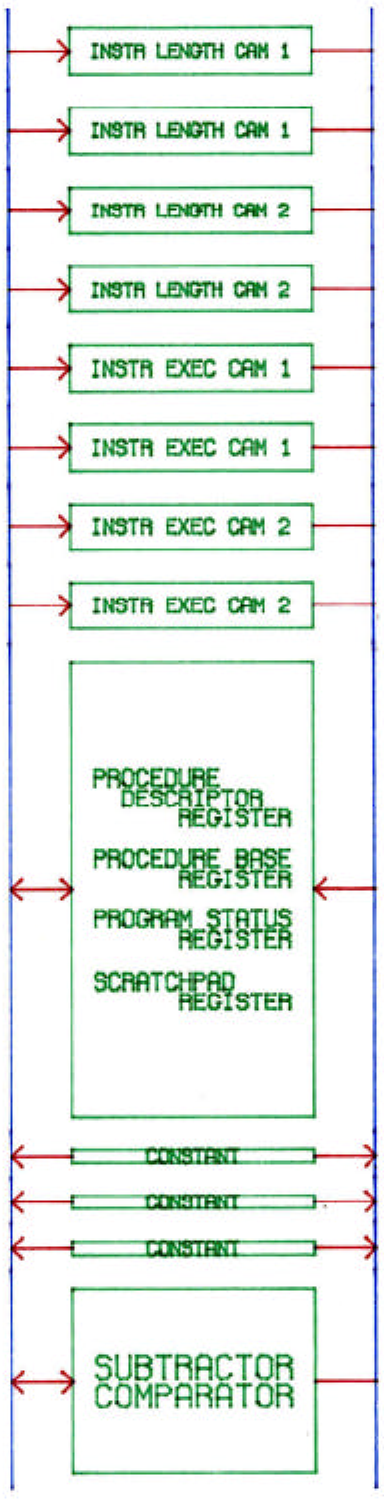


Figure 14a: Instruction Unit Decode Section Block Diagram

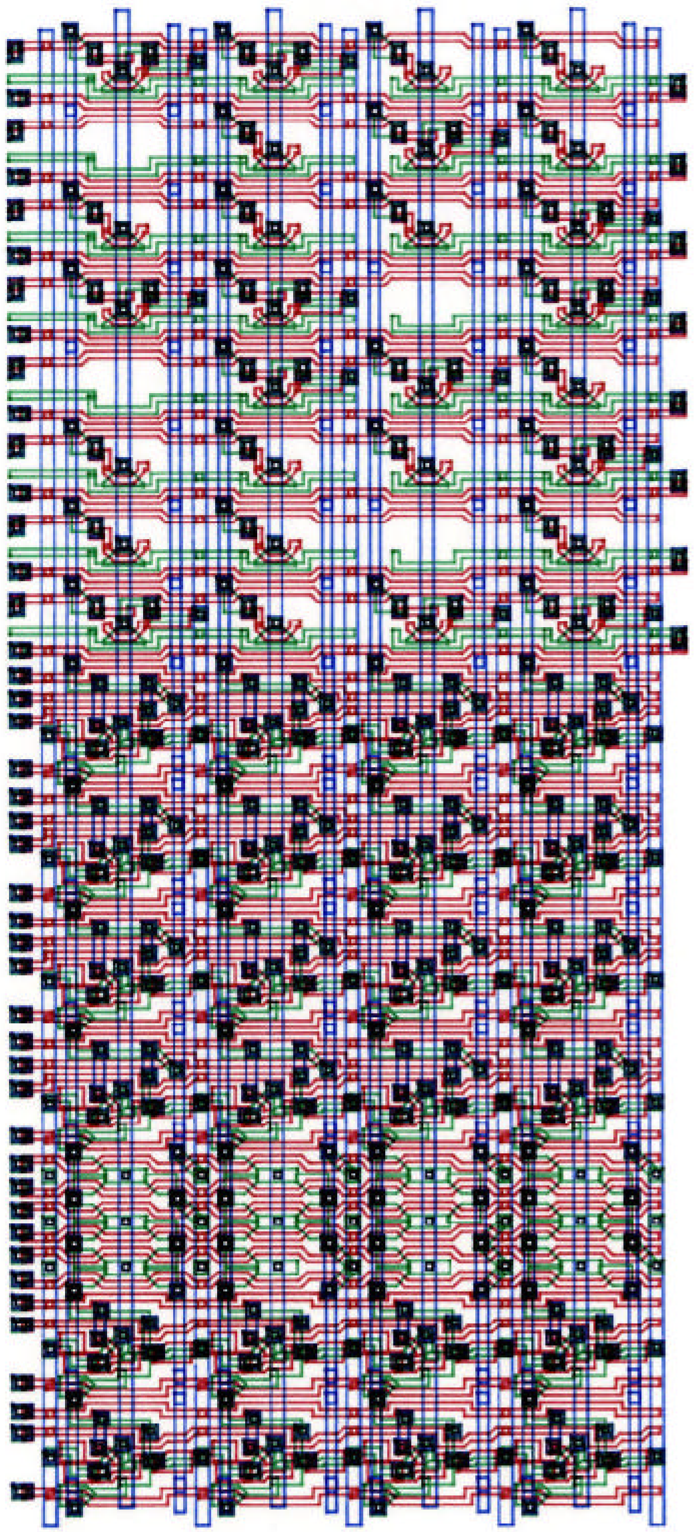


Figure 14b: Instruction Unit Decode Section Layout

9.2 Branch Instructions

There are two kinds of branch instructions: conditional and unconditional. Unconditional forward branches are handled by updating the PC, invalidating the prefetch buffers, and continuing. Short backward branches could be done completely within the instruction unit by saving previously decoded instructions. Tannenbaum's frequency data indicates that backward branches are seldom short. Therefore, unconditional forward and backward branches are handled in the same way. An unconditional branch does not require that any of the instructions issued prior to it finish executing.

A conditional branch, on the other hand, does require that all prior instructions finish executing before it can be resolved. When a conditional branch instruction is decoded, the instruction unit invalidates the prefetch buffers and refills them: one from the present value of the PC, and one from the PC plus the branch displacement. When the completed register matches the accepted register, meaning that all prior instructions have completed execution, the instruction unit is free to read the comparison value(s) from the stack and perform the comparison. It then knows which prefetch buffer to use, so it issues an ACCEPT INSTR message followed by a STACK POP 1 or STACK POP 2 message, depending whether two stack words are being compared or one word is being compared to zero. The instruction unit then continues executing normally.

If greater concurrency is needed, a future version of the Pascal system could employ a more complicated scheme involving the pre-issuing of instructions from both sides of the conditional branch. After the branch has been calculated, a special stack bus message would be sent indicating which set of instructions to start executing. This method would gain a little extra concurrency, but at the cost of moving some of the instruction unit's function into all the other processors. This violates the principal of separation of function, so it is not a good thing to do. The resulting extra complexity makes it doubtful that such a scheme would be useful in any case.

9.3 Procedure Call and Return Mechanism

Procedure calls are very common in most Pascal programs. In Tannenbaum's frequency data, procedure calls accounted for more than 10% of the instructions. Therefore it is important that they be handled efficiently. This is difficult in this architecture, because procedure calls and returns affect the instruction unit, the memory manager and the stack manager. Therefore, they cannot be executed in any single one of the three. The solution used in the Pascal system is to have the instruction unit issue certain special instructions before and after the procedure call and return instructions and to redefine slightly what those instructions do.

A procedure call is executed in the following way. Upon decoding the call instruction (cal n), the instruction unit reads in the procedure descriptor indexed by the instruction operand. The procedure descriptor contains the procedure starting address and its number of words of parameters. The instruction unit loads the PC with the procedure starting address and pushes the old value of the PC onto the stack just as if it had decoded a load constant instruction. It then issues the call instruction, replacing the old operand with the number of words of parameters and follows this by issuing a store local base instruction (str 2). The stack manager accepts the call instruction. When the time comes to execute it, the stack manager pops the return address from the stack and writes it into the space provided in the administration area. The stack manager finds the administration area by subtracting the number of words of parameters from the stack pointer. The stack manager then writes the stack out to main memory, so that the memory manager can read the actual parameter values that were loaded onto the stack. Finally, the stack manager pushes the address of the top of the administration area onto the stack, so that the memory manager can store it into the local base register.

The procedure return instruction is also executed by the stack manager. The stack manager executes this instruction normally, with the exception that it leaves the dynamic link and return address on the top of the stack. After issuing the return instruction, the instruction unit issues store local base (str 2) to restore the local base register, and then waits for all prior instructions to complete so that it can remove the return address from the stack. The return address is removed by issuing the messages ACCEPT INSTR and POP STACK 1.

As defined above, the procedure call instruction does not require the instruction unit to wait for all prior instructions to complete. The return instruction does. If this is shown to be a problem, there is a fairly simple change that can be made within the instruction unit which removes this limitation. When a procedure call is encountered, the present value of the PC is not pushed onto the stack. Instead, it is saved in the return address register. The old value of this register is pushed onto the stack instead. When a return instruction is decoded, the return address register is immediately copied into the PC and instruction prefetch continues. When the return instruction has finished execution and the dynamic link has been removed from the stack, the return address from the administration area is copied into the return address register. The result of this is that the most recent return address is kept in the instruction unit and each administration area contains the return address for the previous procedure.

There is one more complication in the definition of the return address register. The register must never be read while its value is invalid, that is, after a procedure return but before the return instruction has completed execution. This can be assured by writing -1 into the register every time its value is read. Then if a procedure call or return instruction is decoded while the return address register register is still -1, the instruction unit must wait until the previous return instruction has finished execution. It is not likely that this will occur very often.

9.4 Instruction Pre-issue

It is instructive to try to determine the average number of instructions that have been issued across the stack bus but have not yet completed execution. This is equivalent to finding the average difference between the accepted and completed registers. Simulation, or at least some dynamic frequency data, is necessary to get a real answer. A good guess at the upper limit can be reached by using static frequency data and making some simplifying assumptions.

The first assumption is that the static and dynamic frequencies are essentially the same. Data gathered by Tannenbaum for programs written in a simple, structured language strongly supports this assumption.[12] For most instructions that he measured, the dynamic frequency was within 20% of the value of the static frequency. The most variance was shown among instructions with low frequencies.

The second assumption is that procedure returns are the only important exception to the first assumption. Tannenbaum does not compile data on the number of procedure returns, but it is not necessary. The static frequency is roughly equivalent to the number of procedures, and the dynamic frequency is identical to the frequency of procedure calls. Consulting table 3 in appendix A, the static frequency of procedure calls is 10.26%. Therefore it is acceptable to assume that procedure return instructions occur with a dynamic frequency of 10%.

The final assumption is useful only in placing an upper limit on the average number of instructions issued but not yet completed. It is that the instruction unit issues instructions infinitely fast, stopping only when it must wait for all prior instructions to complete. This assumption implies that if the average number of instructions that cause the instruction unit to wait is 1 in *n*, then the average number of instructions issued but not executed is n/2.

The only instructions that cause the instruction unit to wait are the conditional branches and procedure returns. Summing up their frequencies from section 3 in appendix B, we see that 14.9% of the instructions cause the instruction unit to wait. This works out to 1 in every 6.7 instructions, for an average of 3.4 instructions issued but not completed. This number is unfortunately low, especially considering that it is an upper limit.

In the last section we saw that is is possible to implement procedure returns in such a way that they do not require the instruction unit to wait for all prior instructions to complete. If this is done, the frequency of instructions that cause the instruction unit to wait is only 4.6%, or 1 in 22 instructions. This gives 11 as an upper limit for the average number of instructions issued but not completed. From this discussion, it is clear that the return address register should be included in the instruction unit.

10. Conclusions

This thesis has presented a Pascal machine architecture designed using custom VLSI. The design is not complete, but all major aspects of the architecture are specified and alternatives are listed for the areas that are not specified. The main areas yet to be resolved are the design of the processor microcode controllers, the specification of the message busses and their interfaces, and the design of the floating point processor. These are left unspecified primarily because of time constraints. However, the microcode controllers also could not be specified because of a limitation in Bristle Blocks.

The decode section of a Bristle Blocks datapath processor is just the AND plane of a PLA. Minterms can be generated to produce AND/OR functions, but the decoder section contains no clocks and therefore no state. This means that the microcode inputs must either come from off-chip or from a datapath element on-chip. For certain processors in the Pascal system, it is clear that a much simpler control mechanism is sufficient. A state machine, implemented as a clocked PLA, could easily handle the sequencing of the local stack controllers and the integer processor. This would be vastly more efficient than using a microcode controller and off-chip microcode memory.

The present version of Bristle Blocks does not allow the decoder section to be a state machine, but plans have been made to combine Ron Ayres' heirarchical PLA work[3] with Bristle Blocks. Using Ron Ayres' PLA functions, it would be possible to make the decoder section be an arbitrarily complex heirarchy of clocked PLAs. By having some PLA inputs come from off-chip, it would even be possible to mix external microcode with the internal state machine.

There is a great deal of further work that can be done on both the Pascal system and Bristle Blocks. Allowing the decoder section to be a state machine is an important generalization of Bristle Blocks. Also, much work can be done designing new datapath elements and improving the Bristle Blocks user interface. For the Pascal system, the main work involves specifying the message busses, designing an on-chip bus interface, and designing a floating point processor. The message busses could also be optimized on a basis of expected use. The microcode controller design work should wait until the more generalized Bristle Blocks decoder section is implemented.

References

[1] Ayres, Ron, "IC Specification Language", *Proceedings of the 16th Annual Design Automation Conference*, June, 1979

[2] Ayres, Ron, "A Language Processor and a Sample Language", PhD Thesis, California Institute of Technology, 1979

[3] Ayres, Ron, "Silicon Compilation- Heirarchical Use of PLAs", *Proceedings of the 16th Annual Design Automation Conference*, June, 1979

[4] Efland, Greg, "An IC Breadboard Facility Based on the TriMOSbus", MS Thesis, California Institute of Technology (in preparation)

[5] Johannsen, Dave, "Bristle Blocks: A Silicon Compiler", *Proceedings of the Caltech Conference on VLSI*, January 22, 1979

[6] Johannsen, Dave, Bristle Blocks Program Text, Caltech DecSystem-20 files <DAVE>DH*.ICL and <DAVE>DI*.ICL, June, 1979

[7] Mead, Carver, and Sutherland, Ivan, "Micro-electronics and Computer Science", *Scientific American,* Volume 237, Number 3, September, 1977

[8] Mead, Carver and Conway, Lynn, *Introduction to VLSI Systems*, Addison-Wesley, to be published August, 1979

[9] Seiler, Larry, "Transaction Arbitration for the TriMOSbus", Caltech SSP memo # 2163, November 16, 1978

[10] Stevenson, Johan, and Tannenbaum, Andrew, "Efficient Encoding of Machine Instructions", submitted for publication March, 1979

[11] Sutherland, Ivan, Molnar, Charles, Sproull, Robert, and Mudge, Craig, "TriMOSbus ", *Proceedings of the Caltech Conference on VLSI*, January 22, 1979

[12] Tannenbaum, Andrew, "Implications of Structured Programming for Machine Architecture", *Communications of the ACM*, Volume 21, Number 3, March, 1978

[13] Tannenbaum, Andrew, Stevenson, Johan, and van Staveren, Hans, "Description of an Experimental Machine Architecture for Use with Block Structured Languages", unpublished, version of March 2, 1979

Appendix A: Bristle Blocks Datapath Elements

This appendix documents the primitive datapath elements designed by the author to be used in the Pascal system. For each datapath element there is a description, followed by the ICL code which defines it, followed by the layout and circuit diagram representations of the element. The code is kept in four separate files: the runtime file, the details file, the source file, and the test file. The runtime file contains functions which are executed when the chip is generated. The source file contains the code which creates the data file used to generate the element at runtime. The details file contains information used by the runtime and source files. The test file generates an instance of the datapath element which is used to generate the layout and circuit diagram of each datapath element.

A little background will make it easier to read the code files. The entire Bristle Blocks system is coded in ICL (Integrated Circuit Language), a language designed at Caltech by Ron Ayres as part of his Doctoral thesis.[2] Although it was created for the purpose of IC design, it is a general purpose programming language, with features such as strong datatyping, list structures, polymorphic function names, and coercions. Points are defined as a primitive datatype and are entered using the sharp, or pound sign. For example, 0#0 denotes the origin. As in most languages, parentheses ((...)) delimit function parameter lists with commas separating the elements in each list. In ICL, brackets ([...]) delimit record structures, and braces ({...}) delimit strings. Spaces separate the elements in a record structure, and semicolons separate the elements in a string. Single quotes ('...') delimit character strings and double quotes ("...") delimit comments. A backslash (\ ) indicates an infix or postfix function call. The forms <$, $$, and $> are string append and concatenate operators. The form // [...] ... \\ represents a suspendable function. This is a function which is stored as data. The brackets contain variables whose values are passed to the suspendable function. For further information about ICL, see the *ICL Reference Manual*, which is contained in Ron Ayres' PhD thesis.[2]

Several things are useful to know about the Bristle Blocks program code. The three data types which are most important to know about are BLOCK, used to describe datapath elements, VBLOCK, a special type of BLOCK used in the source files, and BLOCK_PRODUCER, which is used in the runtime file. A BLOCK_PRODUCER is a suspendable function which returns a variable of type BLOCK. DATAPATH is the global variable used to produce the datapath. It is a *string of* BLOCK_PRODUCERS. *USPEC_PRODUCER and MEMORY_USPEC are types used to specify* microcode functions. The variable UNUSED is a pre-defined USPEC_PRODUCER which has a null function. This variable is used in the test files, since a microcode function is unnecessary to test plot the datapath.

1. Constant Source

This datapath element is used to source constants onto the data busses. constants are specified either as integers or as character strings representing binary, octal, or hex numbers. Each constant source element has two separate control lines. Each control line can be specified to drive a constant onto either or both busses.

Figure 15 shows three constant source elements. The six control lines drive constants onto the busses as shown in table 10.

Control Line	Upper Bus	Lower Bus
1 left	-	11
1 right	9	-
2 left	7	0
2 right	10	5
3 left	4	-
3 right	14	-

Table 10: Constant Source Values

1.1 Runtime File

```
"CONST.ICL:        Runtime function definitions for constant value source"

/*READ CONSTD;*/     "also requires BINARY.ICL"


"Constant value source:  Description of Parameters"

"        Name:  name of block for block diagram"
"        U1:    microcode spec for driving first  constant onto bus"
"        U2:    microcode spec for driving second constant onto bus"
"        UL:    binary constant to be driven onto upper bus when U1 is true"
"        LL:    binary constant to be driven onto lower bus when U1 is true"
"        UR:    binary constant to be driven onto upper bus when U2 is true"
"        LR:    binary constant to be driven onto lower bus when U2 is true"

VAR  NULL = BINARY;  NULL := NIL;

"        If any of the binary constants has a value of NULL, that bus is not
         driven while that microcode function is enabled.  By using NULL values,
         constants may be sourced to only one of the two busses, allowing
         another microcode driver to source the other bus."
```

```
DEFINE CONSTANT(NAME:SC  U1,U2:USPEC_PRODUCER  UL,LL,UR,LR:BINARY)
                                                          =BLOCK_PRODUCER:
    //  [ NAME; U1; U2; UL; LL; UR; LR; ]
        BEGIN  VAR A, B = BLOCK;  I = INT;
        DO
            B := NEXT_GUY(9,9,5,5,0,8,0/1);
            A := NEXT_BLOCK;

            @(A).CALLS := { B          \AT 14#0;
                            CONST_COL \AT  0#0  \WITH_USPECS {U1;U2};

                            FOR I FROM 0 TO WIDTH-1; COLLECT
                                IF DEFINED(LL) THEN
                                    IF TAIL(LL) THEN CONST_LL_1 \AT 0#I*HEIGHT
                                                ELSE CONST_LL_0 \AT 0#I*HEIGHT
                                    FI
                                ELSE NIL
                                FI;

                            FOR I FROM 0 TO WIDTH-1; COLLECT
                                IF DEFINED(UL) THEN
                                    IF TAIL(UL) THEN CONST_UL_1 \AT 0#I*HEIGHT
                                                ELSE CONST_UL_0 \AT 0#I*HEIGHT
                                    FI
                                ELSE NIL
                                FI;

                            FOR I FROM 0 TO WIDTH-1; COLLECT
                                IF DEFINED(LR) THEN
                                    IF TAIL(LR) THEN CONST_LR_1 \AT 0#I*HEIGHT
                                                ELSE CONST_LR_0 \AT 0#I*HEIGHT
                                    FI
                                ELSE NIL
                                FI;

                            FOR I FROM 0 TO WIDTH-1; COLLECT
                                IF DEFINED(UR) THEN
                                    IF TAIL(UR) THEN CONST_UR_1 \AT 0#I*HEIGHT
                                                ELSE CONST_UR_0 \AT 0#I*HEIGHT
                                    FI
                                ELSE NIL
                                FI
                          };

            @(A).VIEWS := { BLOCK(NAME,14,DEFINED(UL) ! DEFINED(UR),
                              FALSE,FALSE,DEFINED(LL) ! DEFINED(LR)  )   };

        GIVE A END
    \\
ENDDEFN
```

```
DEFINE CONSTANT(NAME:SC  U1,U2:USPEC_PRODUCER  UL,LL,UR,LR:BINARY):
    DATAPATH ::= $> CONSTANT(NAME,U1,U2,UL,LL,UR,LR);
ENDDEFN


"Multiple Constant value source:  Description of Parameters"

"        Name:  name of block for block diagram"
"        U1:    microcode producer for driving constants onto bus"
"        Hi:    binary constants to be driven onto upper bus when U1 is true"
"        Lo:    binary constants to be driven onto lower bus when U1 is true"

DEFINE CONSTANT(NAME:SC  U1:USPEC_PRODUCER  HI,LO:BINARYS):
    BEGIN  VAR HI1,HI2,LO1,LO2 = BINARY; I = INT;
        I := 1;
        WHILE  DEFINED(HI[I]) ! DEFINED(LO[I]);  DO
            CONSTANT(NAME,U1,U1,DEF(HI[I]),DEF(LO[I]),
                                DEF(HI[I+1]),DEF(LO[I+1])    );
            I := I+2;
        END
    END
ENDDEFN


"End of CONST.ICL"
```

1.2 Details File

```
"CONSTD.ICL:        Details for constant value source"

VAR        CONST_NULL, CONST_CONN, CONST_COL              = BLOCK;
VAR        CONST_LL_0, CONST_UL_0, CONST_LR_0, CONST_UR_0 = BLOCK;
VAR        CONST_LL_1, CONST_UL_1, CONST_LR_1, CONST_UR_1 = BLOCK;

"PARAMETERS TO DISK_BLOCK ARE FILE NAME, POSITION IN FILE, UNIQUE ID"

CONST_NULL := DISK_BLOCK('CONST', 1, 3181);
CONST_CONN := DISK_BLOCK('CONST', 2, 3182);
CONST_COL  := DISK_BLOCK('CONST', 3, 3183);
CONST_LL_0 := DISK_BLOCK('CONST', 4, 3184);
CONST_UL_0 := DISK_BLOCK('CONST', 5, 3185);
CONST_LR_0 := DISK_BLOCK('CONST', 6, 3186);
CONST_UR_0 := DISK_BLOCK('CONST', 7, 3187);
CONST_LL_1 := DISK_BLOCK('CONST', 8, 3188);
CONST_UL_1 := DISK_BLOCK('CONST', 9, 3189);
CONST_LR_1 := DISK_BLOCK('CONST',10, 3190);
CONST_UR_1 := DISK_BLOCK('CONST',11, 3191);

"End of CONSTD.ICL"
```

1.3 Source File

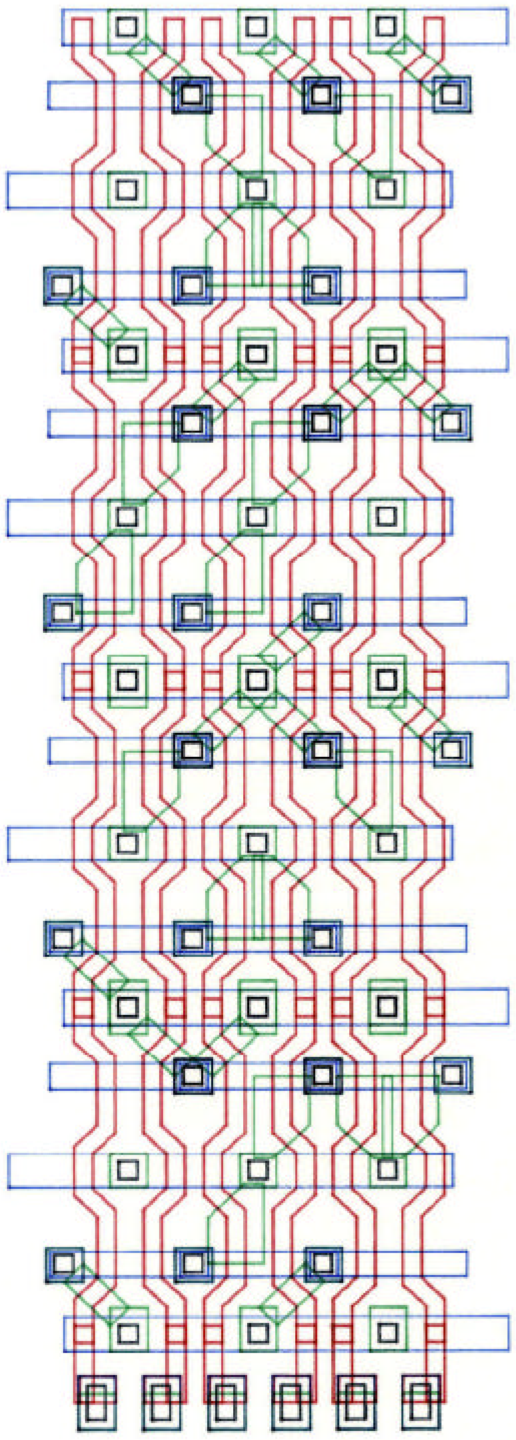
```
"CONSTS.ICL:            Definitions for constant value source"

/*READ CONSTD;*/

VAR         CONST_VNULL, CONST_VCONN, CONST_VCOL              = VBLOCK;
VAR         CONST_VLL_0, CONST_VUL_0, CONST_VLR_0, CONST_VUR_0 = VBLOCK;
VAR         CONST_VLL_1, CONST_VUL_1, CONST_VLR_1, CONST_VUR_1 = VBLOCK;


CONST_VNULL       := NIL;

CONST_VNULL.VIEWS :=
    { LAYOUT ({
        BW ({ 0#0\Y2; 14#. });                   "lower bus"
        BW ({ 0#0 \Y1; 14#. });                   "upper bus"

        RW ({ 2#0\Y4; .#-4.5\Y2; 4.5#.+2.5;
              .#.+8; 2#.+2.5; 2#-8.5 \Y1;
              4.5#.+2.5; .#.+8; 2#.+2.5;
                                   .#0\y3   }); "right control wire"
        RW ({ 12#0\Y4; .#-4.5\Y2; 9.5#.+2.5;
              .#.+8; 12#.+2.5; 12#-8.5 \Y1;
              9.5#.+2.5; .#.+8; 12#.+2.5;
                                   .#0\y3   }); "left  control wire"

        GB ( 5#-2.0\Y4, 9#-5.0\Y2 );             "lower VDD connect"
        GB ( 5#-9.0 \Y1, 9#9\Y2    );             "GND connect"
        GB ( 5#2\y3,     9#5 \Y1    );             "upper VDD connect"

        KB ( 6#1\Y4, 8#-1.0\Y4 );                "lower VDD contact cut"
        KB ( 6#1,    8#-1.0    );                "GND contact cut"
        KB ( 6#1\y3, 8#-1.0\y3 )                 "upper VDD contact cut"
    }) };


CONST_VLL_0       := NIL;

CONST_VLL_0.CALLS := { GCB \AT 0#0\Y2 };        "lower bus contact"

CONST_VLL_0.VIEWS :=
    { LAYOUT ({
        GP ({ 1.5#0\Y2; 7.5#.; .#.+9.5;
                        5.5#.; 1.5#.-4.0 })     "pulldown path"
    }) };


CONST_VUL_0       := NIL;

CONST_VUL_0.CALLS := { GCB \AT 0#0 \Y1 };        "upper bus contact"

CONST_VUL_0.VIEWS :=
    { LAYOUT ({
        GP ({ 1.5#0 \Y1; 7.5#.; .#.-9.5;
                        5.5#.; 1.5#.+4  })      "pulldown path"
    }) };


CONST_VLR_0       := NIL;

CONST_VLR_0.CALLS := { CONST_LL_0 \MIRY \AT 14#0 };
```

```
CONST_VUR_0        := NIL;

CONST_VUR_0.CALLS := { CONST_UL_0 \MIRY \AT 14#0 };


CONST_VLL_1        := NIL;

CONST_VLL_1.CALLS := { GCB \AT 0#0\Y2 };          "lower bus contact"

CONST_VLL_1.VIEWS :=
    { LAYOUT ({
        GP ({ 2#0\Y2; 7#.-5; 5#.-2; 0#.+5 })     "pullup path"
    }) };


CONST_VUL_1        := NIL;

CONST_VUL_1.CALLS := { GCB \AT 0#0 \Y1 };         "upper bus contact"

CONST_VUL_1.VIEWS :=
    { LAYOUT ({
        GP ({ 2#0 \Y1; 7#.+5; 5#.+2; 0#.-5 })    "pullup path"
    }) };


CONST_VLR_1        := NIL;

CONST_VLR_1.CALLS := { CONST_LL_1 \MIRY \AT 14#0 };


CONST_VUR_1        := NIL;

CONST_VUR_1.CALLS := { CONST_UL_1 \MIRY \AT 14#0 };


CONST_VCONN        := NIL;

CONST_VCONN.CALLS := { GRCBU \AT { 3.5#1\Y6; 10.5#1\Y6 } };

CONST_VCONN.VIEWS :=
    { LAYOUT ({
        RW ({  2#0\Y4;  2#3\Y6 });               "left control connect"
        RW ({ 12#0\Y4; 12#3\Y6 })                "right control connect"
    }) };


CONST_VCOL := NIL;

CONST_VCOL.CALLS := { CONST_CONN;
                       CONST_NULL \AT EACH_BIT_POSITION };

CONST_VCOL.INTERFACE :=
    { [ FROM: 3.5#0\Y6  TYPE:1  BUFFER:1  EDGE:3  UCODE:1 ];
      [ FROM:10.5#0\Y6  TYPE:1  BUFFER:1  EDGE:3  UCODE:1 ]  };


SCRATCH_FILE(CONST_NULL.DETAILS.FILE_NAME);
```

```
DUMP_SET({  [ B:CONST_NULL  V:CONST_VNULL ];
            [ B:CONST_CONN  V:CONST_VCONN ];
            [ B:CONST_COL   V:CONST_VCOL  ];
            [ B:CONST_LL_0  V:CONST_VLL_0 ];
            [ B:CONST_UL_0  V:CONST_VUL_0 ];
            [ B:CONST_LR_0  V:CONST_VLR_0 ];
            [ B:CONST_UR_0  V:CONST_VUR_0 ];
            [ B:CONST_LL_1  V:CONST_VLL_1 ];
            [ B:CONST_UL_1  V:CONST_VUL_1 ];
            [ B:CONST_LR_1  V:CONST_VLR_1 ];
            [ B:CONST_UR_1  V:CONST_VUR_1 ]  });

"End of CONSTS.ICL"
```

1.4 Test File

```
"CONSTT.ICL:          Tests the constant value source"


VAR  A = BLOCK:

PRECHARGE := FALSE;
DATA_WIDTH(4);


CONSTANT('Const', UNUSED, UNUSED,  NULL, 'H B', 'Q11',  NULL );
CONSTANT('Const', UNUSED, UNUSED, 'H 7',    0 ,   10 , '101' );
CONSTANT('Const', UNUSED, UNUSED, '100',  NULL, 'Q16',  NULL );

A := DATAPATH;

"End of CONSTT.ICL"
```



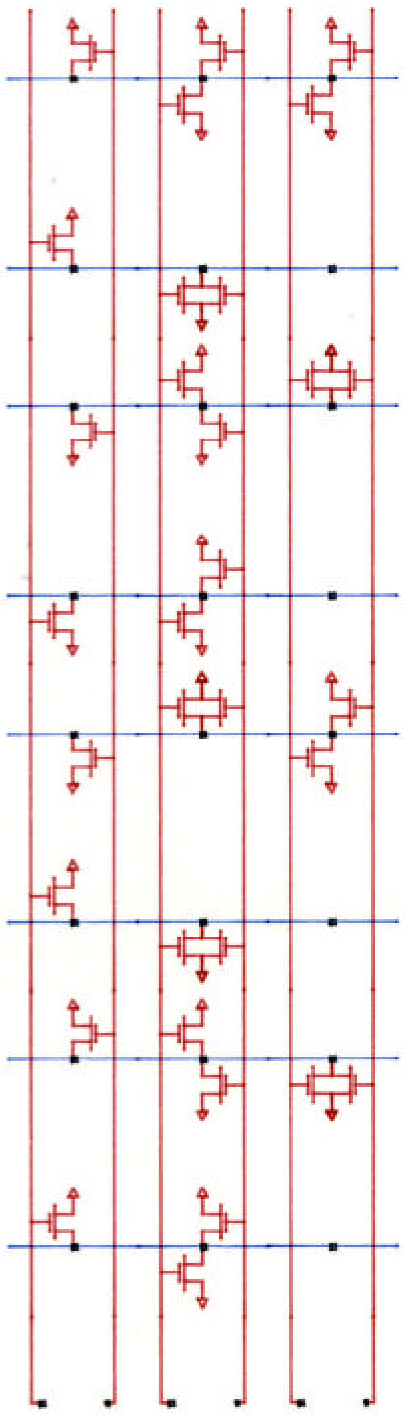


Figure 15: Constant Source Layout and Circuit Diagram

2. Fixed Value Content Addressable Memory

This datapath element compares a value loaded from the lower data bus against a fixed value. If they match, a compare out line goes high. In the Pascal system, these elements are used to recognize memory bus message headers.

Figure 16 shows two fixed value CAMs. Each has a compare output line and a control line for loading the word to be compared. The first CAM compares against the binary value '0011'. The second CAM compares against the value '1XX0', that is, it compares the high order bit against 1, ignores the middle two bits and compares the low order bit against 0.

2.1 Runtime File

```
"CAM1.ICL:          Runtime function definitions for content addressable
                    memories with fixed mask and fixed compare value     "

/*READ CAM1D;*/       "also requires BINARY.ICL"


"Fixed value CAM with no Masking:  Description of Parameters"

"         Name:  name of block for block diagram"
"         U1:    microcode spec for loading word to be compared"
"         U2:    microcode spec for comparison output"
"         Value: binary constant indicating the comparison value"

DEFINE CAM(NAME:SC  U1,U2:USPEC_PRODUCER  VALUE:BINARY)=BLOCK_PRODUCER:
    //  [ NAME; U1; U2; VALUE; ]
        BEGIN  VAR A, B = BLOCK;  I = INT;
        DO
            B := NEXT_GUY(12,12,5,5,0,6,1/6);
            A := NEXT_BLOCK;

            @(A).CALLS := { B        \AT 21#0;
                            CAM1_COL \AT  0#0   \WITH_USPECS {U1;U2};

                            FOR I FROM 0 TO WIDTH-1; COLLECT
                                IF TAIL(VALUE) THEN CAM1_ONE \AT 0#I*HEIGHT
                                               ELSE CAM1_ZERO\AT 0#I*HEIGHT
                                FI
                          };

            @(A).VIEWS := { BLOCK(NAME,21,FALSE,FALSE,TRUE,FALSE) };

        GIVE A END
    \\
ENDDEFN

DEFINE CAM(NAME:SC  U1,U2:USPEC_PRODUCER  VALUE:BINARY):
    DATAPATH ::= $> CAM(NAME,U1,U2,VALUE);
ENDDEFN
```

```
"Fixed value CAM with Fixed Mask:  Description of Parameters"

"         Name:  Name of block for block diagram"
"         U1:    Microcode spec for loading word to be compared"
"         U2:    Microcode spec for comparison output"
"         Mask:  Binary constant indicating which bits are compared to 1,"
"                which bits are compared to 0, and which bits are masked."

DEFINE CAM(NAME:SC  U1,U2:USPEC_PRODUCER  MASK:MASK)=BLOCK_PRODUCER:
    //  [ NAME; U1; U2; MASK; ]
        BEGIN  VAR A, B = BLOCK; I = INT;
        DO
            B := NEXT_GUY(12,12,5,5,0,6,1/6);
            A := NEXT_BLOCK;


            @(A).CALLS := { B        \AT 21#0;
                            CAM1_COL \AT  0#0  \WITH_USPECS {U1;U2};

                            FOR I FROM 0 TO WIDTH-1; COLLECT
                                CASE TAIL(MASK) OF
                                    NONE:  NIL
                                    ZERO:  CAM1_ZERO \AT 0#I*HEIGHT
                                    ONE:   CAM1_ONE  \AT 0#I*HEIGHT
                                ENDCASE
                          };

            @(A).VIEWS := ( BLOCK(NAME,21,FALSE,FALSE,TRUE,FALSE) );
        GIVE A END
    \\
ENDDEFN

DEFINE CAM(NAME:SC  U1,U2:USPEC_PRODUCER  MASK:MASK):
    DATAPATH ::= $> CAM(NAME,U1,U2,MASK);
ENDDEFN

"End of CAM1.ICL"
```

2.2 Details File

```
"CAM1D.ICL:        Details for content addressable memories
                   with fixed mask and fixed compare value  "

VAR     CAM1_NULL, CAM1_NUM, CAM1_ZERO, CAM1_ONE, CAM1_CONN, CAM1_COL  = BLOCK;

"PARAMETERS TO DISK_BLOCK ARE FILE NAME, POSITION IN FILE, UNIQUE ID"

CAM1_NULL := DISK_BLOCK('CAM1', 1, 3681);
CAM1_NUM  := DISK_BLOCK('CAM1', 2, 3682);
CAM1_ZERO := DISK_BLOCK('CAM1', 3, 3683);
CAM1_ONE  := DISK_BLOCK('CAM1', 4, 3684);
CAM1_CONN := DISK_BLOCK('CAM1', 5, 3685);
CAM1_COL  := DISK_BLOCK('CAM1', 6, 3686);

"End of CAM1D.ICL"
```

2.3 Source File

```
"CAM1S.ICL:         Definitions for content addressable memories
                    with fixed mask and fixed compare value      "

/*READ CAM1D;*/


VAR        CAM1_VNULL, CAM1_VNUM, CAM1_VZERO, CAM1_VONE,
                                  CAM1_VCONN, CAM1_VCOL  = VBLOCK;


CAM1_VNULL       := NIL;                "Contains bus wires & control lines"

CAM1_VNULL.VIEWS :=  { LAYOUT ({

        BW ({ 0#0\Y1; 21#.       });            "upper data bus wire"
        BW ({ 0#0\Y2: 21#.       }):            "lower data bus wire"

        RW ({ 4#0\Y4; .#2.5\Y2; .-2#.+2;
              .#-4.5\Y1; .+2#.+2; .#0\Y3 });    "data load line"

        GW ({ 15#0\Y4; .#6\Y2; 18.5#.;
              .#-6\Y1; 15#.; .#0\Y3      })     "compare line"
    }) };


CAM1_VNUM        := NIL;                "Defines basic comparison cell"

CAM1_VNUM.CALLS  :=
    {
        GCB   \AT 0#0\Y2;       "data load contact"
        GRCBR \AT 6#7\Y2;       "internal data contact"
        GCB   \AT 12#0          "contact to GND"
    };


CAM1_VNUM.VIEWS  := { LAYOUT ({

        RW ({ 12#5; 13.5#.; 16#2.5;
              16#-2.5; 13.5#-5.0; 13#. });      "compare pulldown gate"

        GP ({ 13#2; 17#6; 18#6;
              .#-6.0; 17#.; 13#-2.0    });      "compare pulldown path"
        GP ({ 1#2\Y2; 5#6\Y2; 8#.;
                          8#0\Y2; 1#.  })       "data load path"
    }) };


CAM1_VZERO       := NIL;

CAM1_VZERO.CALLS := { CAM1_NUM };

CAM1_VZERO.VIEWS := { LAYOUT ({
        RW ({ 8#9\Y2; .#-5.0; 12#. }) }) };     "internal  connect"


CAM1_VONE        := NIL;

CAM1_VONE.CALLS  := { CAM1_NUM;
                     GRCBR \AT 6#-7.0\Y1;       "inverter contact"
                     GCB   \AT 9#0\Y3           "VDD contact"
```

```
                   };

CAM1_VONE.VIEWS  := { LAYOUT ({

        RW ({ 8#9\Y2; .#2 });                   "inverter pulldown gate"
        RW ({ 12#-9\Y1; .#5 });                 "compare line gate connect"

        GW ({ 10#0; 5#0; .#-9.0\Y1    });       "inverter pulldown path"
        GW ({ 8#-6.0\Y1; 10#.; .#0\Y3 });       "inverter pullup path"

        RB ( 13#-9.0\Y1, 7#4\Y1 }               "inverter pullup gate"
    }) };


CAM1_VCONN       := NIL;            "Defines connections at bottom of column"

CAM1_VCONN.CALLS := { GRCBL \AT 7#1\Y6    };

CAM1_VCONN.VIEWS := { LAYOUT ({ RW({  4#3\Y6; .#0\Y4 });  "data load connect"
                                 GW({ 15#0\Y6; .#0\Y4 })   "compare line connect"
                                }) };

CAM1_VCONN.INTERFACE := { [ FROM: 8#0\Y6 TYPE:1 BUFFER:1 EDGE:3 UCODE:1 ];
                          [ FROM:15#0\Y6 TYPE:1 BUFFER:3 EDGE:3 UCODE:2 ] };


CAM1_VCOL        := NIL;

CAM1_VCOL.CALLS  := {  CAM1_CONN  PASSING_USPECS {1;2};
                       CAM1_NULL \AT EACH_BIT_POSITION  };


SCRATCH_FILE(CAM1_NULL.DETAILS.FILE_NAME);


DUMP_SET({  [ B: CAM1_NULL   V: CAM1_VNULL ];
            [ B: CAM1_NUM    V: CAM1_VNUM  ];
            [ B: CAM1_ZERO   V: CAM1_VZERO ];
            [ B: CAM1_ONE    V: CAM1_VONE  ];
            [ B: CAM1_CONN   V: CAM1_VCONN ];
            [ B: CAM1_COL    V: CAM1_VCOL  ]
        });

"End of CAM1S.ICL"
```

2.4 Test File

```
"CAM1T.ICL:        Tests content addressable memoruie
                   with fixed mask and fixed compare value"


VAR  A = BLOCK;

PRECHARGE := FALSE;
DATA_WIDTH(4);

CAM( 'Cam', UNUSED, UNUSED, '0011' );
CAM( 'Cam', UNUSED, UNUSED, '1XX0' );

A := DATAPATH;


"End of CAM1T.ICL"
```

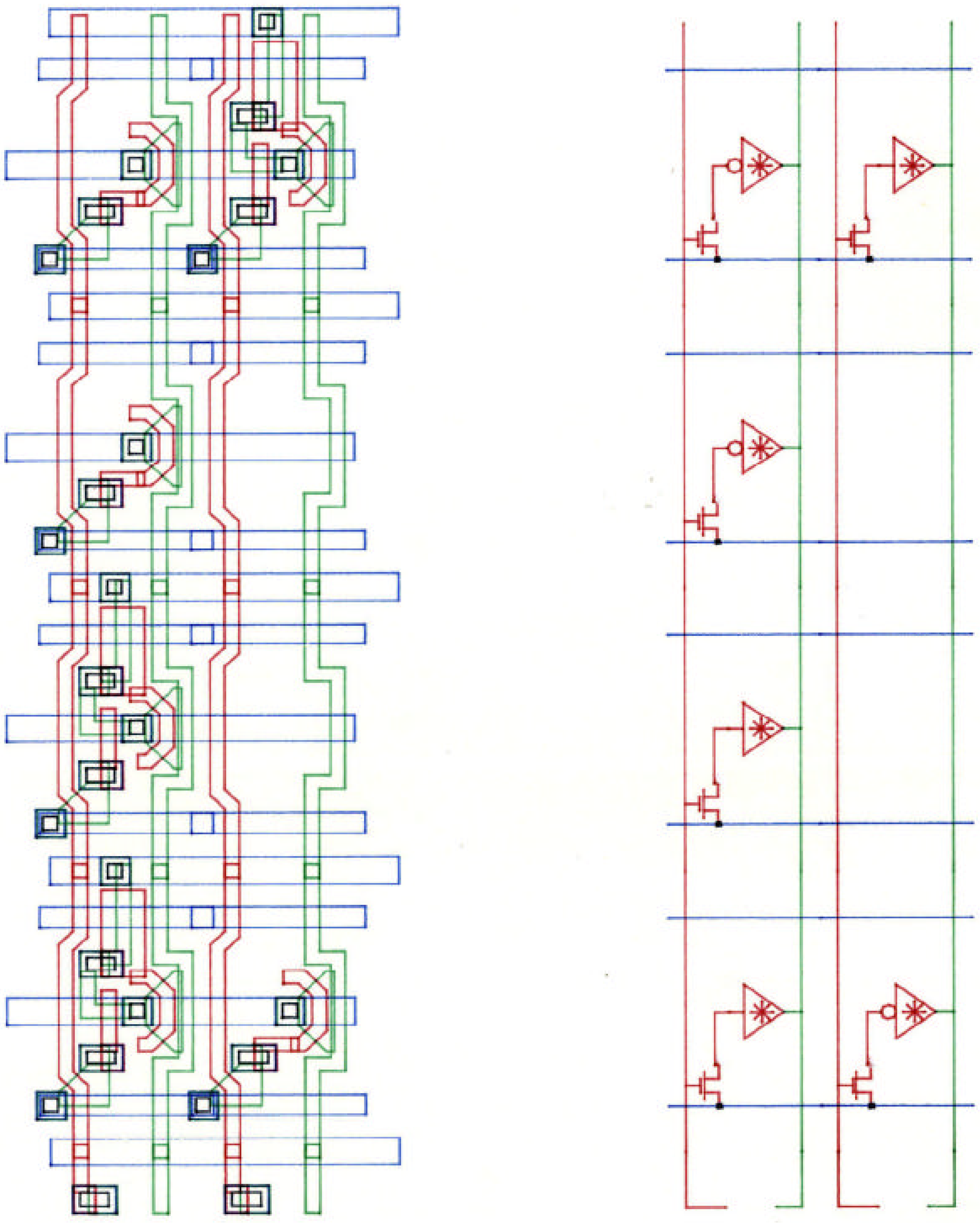


Figure 16: Fixed Value CAM Layout and Circuit Diagram

3. Variable Value Content Addressable Memory

This datapath element compares a value loaded from the lower data bus against a value stored in a memory register. If they match, a compare out line goes high. In the Pascal system, these elements are used to compare the local stack controller's COMPLETED register against the saved value of the ACCEPTED register. They could also be used in place of the fixed value CAMs, if the memory bus message heaaders were not specified prior to chip fabrication.

Figure 17 shows two variable value CAMs. The first compares against all four bits. The second does not compare against bits 1 and 3. The value the bus is compared against is stored in a standard memory register. The first CAM's memory register can be written from both busses and read onto the lower bus. The second CAM's memory register can be read from both busses and written onto the upper bus. Both memory registers have refresh control lines (not shown in the circuit diagram). Normally these are driven by the Ø1 clock.

3.1 Runtime File

```
"CAM2.ICL:          Runtime function definitions for content addressable
                    memories with fixed mask and variable compare value   "

/*READ CAM2D-L01D;*/       "also requires BINARY.ICL"


"Variable value CAM with no Masking:  Description of Parameters"

"        Name:  name of block for block diagram"
"        U1:    microcode spec for loading word to be compared"
"        U2:    microcode spec for comparison output"
"        MU:    memory microcode specs for comparison register"

DEFINE CAM(NAME:SC  U1,U2:USPEC_PRODUCER  MU:MEMORY_USPEC)=BLOCK_PRODUCER:
    //  [ NAME; U1; U2; MU; ]
        BEGIN  VAR A, B = BLOCK; I = INT; MR=MEMORY_RETURN;
        DO
            MR := MEMORY_REGISTER(MU,[A:0 B:0],[A:1 B:0],[A:0 B:0],[A:0 B:0]);
            B  := NEXT_GUY(14,14,5,5,0,8,1/8+1/4+MR.POWER);
            A  := NEXT_BLOCK;

            @(A).CALLS := { B\AT 31+MR.WIDTH#0;
                            CAM2_ALL\AT MR.WIDTH#0 \WITH_USPECS {U1;U2};
                            MR.CALLS
                          };

            @(A).VIEWS := { BLOCK(NAME,31+MR.WIDTH,DEFINED(MU.WU),
                                        DEFINED(MU.RU),TRUE,TRUE  ) };
        GIVE A END
    \\
ENDDEFN

DEFINE CAM(NAME:SC  U1,U2:USPEC_PRODUCER  MU:MEMORY_USPEC):
    DATAPATH ::= $> CAM(NAME,U1,U2,MU);
```

```
ENDDEFN


"Variable value CAM with Fixed Mask:  Description of Parameters"

"         Name:  Name of block for block diagram"
"         U1:    Microcode spec for loading word to be compared"
"         U2:    Microcode spec for comparison output"
"         MU:    Microcode specs for comparison register"
"         Mask:  Binary constant indicating which bits are compared.
                 Bits marked zero are ignored in the comparison."

DEFINE CAM(NAME:SC  U1,U2:USPEC_PRODUCER  MU:MEMORY_USPEC
                                            MASK:BINARY     ) = BLOCK_PRODUCER:
    //  [ NAME; U1; U2; MU; MASK; ]
        BEGIN  VAR A, B = BLOCK; I = INT; MR = MEMORY_RETURN;
        DO
            MR := MEMORY_REGISTER(MU,[A:0 B:0],[A:1 B:0],[A:0 B:0],[A:0 B:0]);
            B  := NEXT_GUY(14,14,5,5,0,8,1/8+1/4+MR.POWER);
            A  := NEXT_BLOCK;

            @(A).CALLS := { B        \AT 31+MR.WIDTH#0;
                            CAM2_COL \AT MR.WIDTH#0   \WITH_USPECS {U1;U2};

                            FOR I FROM 0 TO WIDTH-1; COLLECT
                                IF TAIL(MASK) THEN CAM2_CELL \AT MR.WIDTH
                                                                  #I*HEIGHT
                                              ELSE NIL
                                FI
                          };

            @(A).VIEWS := { BLOCK(NAME,31+MR.WIDTH,DEFINED(MU.WU),
                                       DEFINED(MU.RU),TRUE,DEFINED(MU.RL) ) };
        GIVE A END
    \\
ENDDEFN

DEFINE CAM(NAME:SC  U1,U2:USPEC_PRODUCER  MU:MEMORY_USPEC  MASK:BINARY):
    DATAPATH ::= $> CAM(NAME,U1,U2,MU,MASK);
ENDDEFN


"End of CAM2.ICL"
```

3.2 Details File

```
"CAM2D.ICL:          Details for content addressable memories
                     with fixed mask and variable compare value"

VAR        CAM2_NULL, CAM2_CELL, CAM2_CONN, CAM2_COL, CAM2_ALL = BLOCK;

"PARAMETERS TO DISK_BLOCK ARE FILE NAME, POSITION IN FILE, UNIQUE ID"

CAM2_NULL := DISK_BLOCK('CAM2', 1, 7681);
CAM2_CELL := DISK_BLOCK('CAM2', 2, 7682);
CAM2_CONN := DISK_BLOCK('CAM2', 3, 7683);
CAM2_COL  := DISK_BLOCK('CAM2', 4, 7684);
CAM2_ALL  := DISK_BLOCK('CAM2', 5, 7685);

"End of CAM2D.ICL"
```

3.3 Source File

```
"CAM2S.ICL:         Definitions for content addressable memories
                    with fixed mask and variable compare value   "

/*READ CAM2D;*/


VAR        CAM2_VNULL, CAM2_VCELL, CAM2_VCONN, CAM2_VCOL, CAM2_VALL = VBLOCK;


CAM2_VNULL        := NIL;                   "Contains bus wires & control lines"

CAM2_VNULL.VIEWS :=  { LAYOUT ({

        BW ({ 0#0\Y1; 31#.        });    "upper data bus wire"
        BW ({ 0#0\Y2; 31#.        });    "lower data bus wire"

        RW ({ 27#0\Y4; .#2.5\Y2;
              29#.+2; .#-4.5\Y1;
              27#.+2; .#0\Y3      });    "data load control line"

        GW ({ 6#0\Y4; .#3.5\Y2;
              1.5#.+4.5; .#-8\Y1;
              6#.+4.5; 6#0\Y3     })     "equal line"
    }) };


CAM2_VCELL        := NIL;               "Defines basic comparison cell"

CAM2_VCELL.CALLS :=
    {
        GCB   \AT { 17#0\Y4;            "VDD contact for ex-or pullup"
                    17#0\Y3;            "VDD contact for inverter pullup"
                    31#0\Y2;            "contact to lower data bus"
                     8#-4.0;            "GND contact for equal line"
                    25#-2.0    };       "GND contact for inverter"
        GRCBL \AT { 24#7\Y2    };       "pulldown contact for inverter"
        GRCBU \AT { 23#-9.0\Y1 };       "pullup contact for inverter"
        GRCBR \AT { 14#-8.0\Y1 };       "register gate contact"
        GRCBD \AT { 15#9\Y2    }        "ex-or pullup contact"
    };


CAM2_VCELL.VIEWS :=  { LAYOUT ({

        BW ({ -3.0#-8.0\Y1; 11#.      });   "extra bus connection"

        RW ({ 21#9\Y2; .#0.5;
                      22.5#2; 25#.    });   "inverter pulldown gate"
        RW ({ 13#-3.0\Y1; 10#.; .#.-8 });   "ex-or gate: inverter"
        RW ({ 17#-9\Y1; .#0           });   "ex-or gate: register"
        RW ({ 11#5\Y2; .#-8; 5.5#.;
              4#.+1.5; .#-1.5; 5.5#0  });   "equal line pulldown gate"
        RW ({ 23#-4.0\Y1; .#.-1       });   "inverter pullup connect"

        GW ({ 21#-10\Y1; 20#.;
              .#0\Y1; 16#.; .#-1\Y3   });   "inverter pullup path"
        GW ({ 14#10\Y2; .#2; 8#.; 7#3;
              .#-9.5\Y1; 7.5#.+.5; 12#. }); "ex-or pulldown path: register"
        GW ({ 14#2; 19#.;
                    20#3; .#-10\Y1    });   "ex-or pulldown path: inverter"
        GW ({ 14#7\Y2; .#.-6; 11#.;
              .#.-7; .+5#.; .#0\Y4     });  "ex-or pullup path"

        GP ({ 25#0; 23#.-2; 19#2; 21#4 });  "inverter pulldown path"
        GP ({ 30#1\Y2; 25#.+5;
              24#.; .#.-6; 30#.        });  "data load path"
        GP ({ 2#-9; 3#.; 7#-5;
                    .#-3; 3#1; 2#.     });  "equal line pulldown path"

        BB ( 6#-6.0,     10#2     );        "equal line GND connect"

        RB ( 13#-4.0\Y1, 24#3\Y1 );         "inverter pullup gate"
        RB ( 17#-2.0\Y2, 11#8\Y2 );         "upper part of ex-or pullup gate"
        RB ( 14#-9.0\Y2, 8#4\Y2  );         "lower part of ex-or pullup gate"

        YB ( 13.5#-5.5\Y1, 23.5#4.5\Y1 );   "upper part inverter pullup implant"
        YB ( 19.5#-9.5\Y1, 23.5#4.5\Y1 );   "lower part inverter pullup implant"
        YB ( 11.5#-1.5\Y2, 16.5#9.5\Y2 );   "upper part of ex-or pullup implant"
        YB ( 8.5#-8.5\Y2,  15.5#3.5\Y2 )    "lower part of ex-or pullup implant"
    } ) };


CAM2_VCONN       := NIL;            "Defines connections at bottom of column"

CAM2_VCONN.CALLS := { GRCBU \AT 25#1\Y6    };

CAM2_VCONN.VIEWS := { LAYOUT ({ GW({  6#0\Y6; .#0\Y4   });
                                RW({ 27#0\Y4; .#2\Y6   })
                              }) };

CAM2_VCONN.INTERFACE := { [ FROM:25#0\Y6 TYPE:1 BUFFER:1 EDGE:3 UCODE:1 ];
                          [ FROM:6#0\Y6  TYPE:1 BUFFER:3 EDGE:3 UCODE:2 ] };


CAM2_VCOL       := NIL;            "Defines a CAM without any comparison cells"

CAM2_VCOL.CALLS := { CAM2_NULL    \AT 0#0     \AT EACH_BIT_POSITION;
                     CAM2_CONN    \AT 0#0      PASSING_USPECS {1;2}  };


CAM2_VALL       := NIL;            "Defines a CAM with comp cells for each bit"

CAM2_VALL.CALLS := { CAM2_COL   PASSING_USPECS (1;2);
                     CAM2_CELL \AT  EACH_BIT_POSITION };


SCRATCH_FILE(CAM2_CELL.DETAILS.FILE_NAME);

DUMP_SET({  [ B: CAM2_NULL    V: CAM2_VNULL ];
            [ B: CAM2_CELL    V: CAM2_VCELL ];
            [ B: CAM2_CONN    V: CAM2_VCONN ];
            [ B: CAM2_COL     V: CAM2_VCOL  ];
            [ B: CAM2_ALL     V: CAM2_VALL  ]
        });

"End of CAM2S.ICL"
```

3.4 Test File

```
"CAM2T.ICL:          Tests content addressable memories with
                     fixed mask and variable compare value    "


VAR  A = BLOCK;

PRECHARGE := FALSE;
DATA_WIDTH(4);


CAM( 'Cam', UNUSED, UNUSED, [RU:UNUSED RL:UNUSED WL:UNUSED C:UNUSED]        );
CAM( 'Cam', UNUSED, UNUSED, [WU:UNUSED RU:UNUSED RL:UNUSED C:UNUSED], '101' );

A := DATAPATH;


"End of CAM2T.ICL"
```

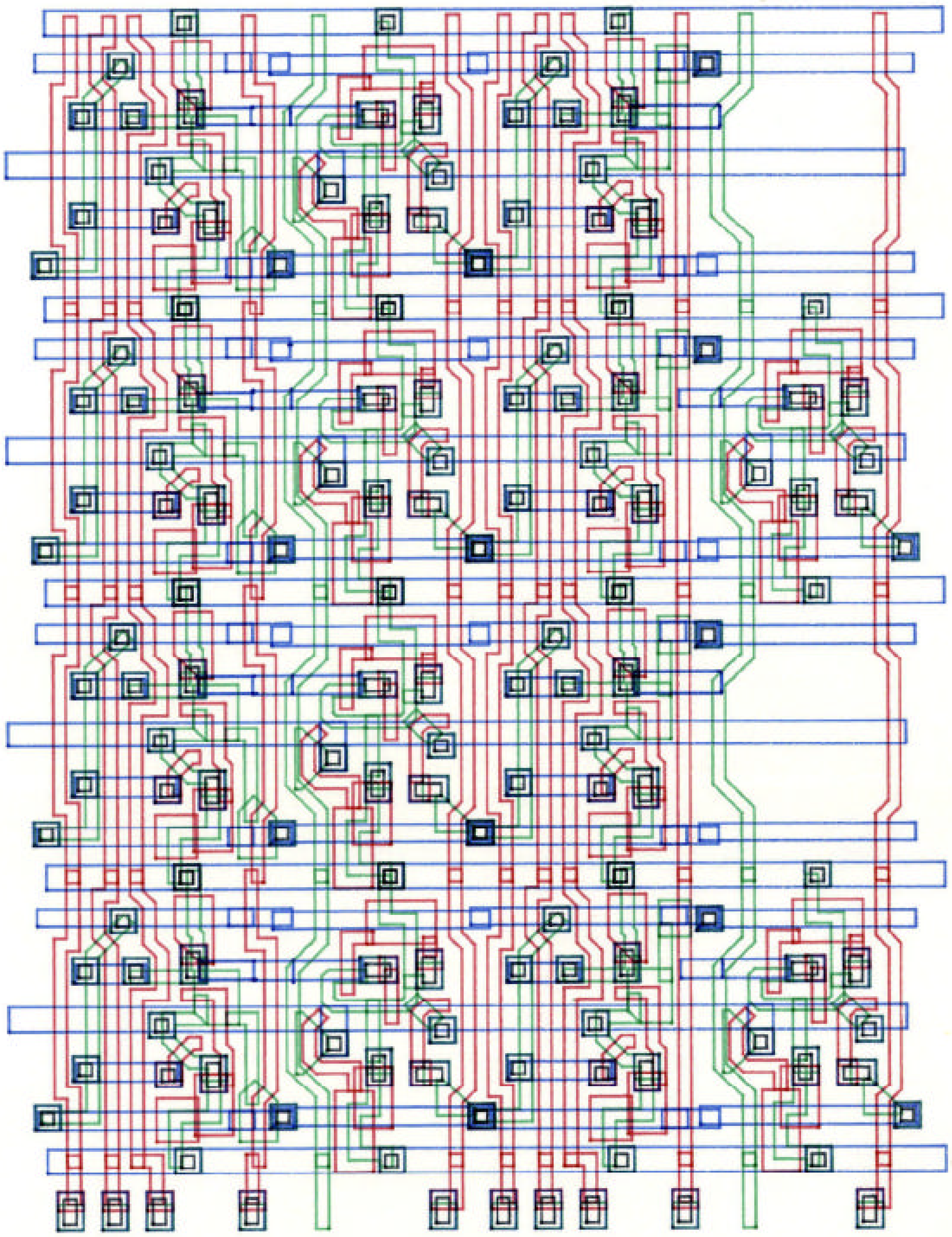


Figure 17a: Variable Value CAM Layout

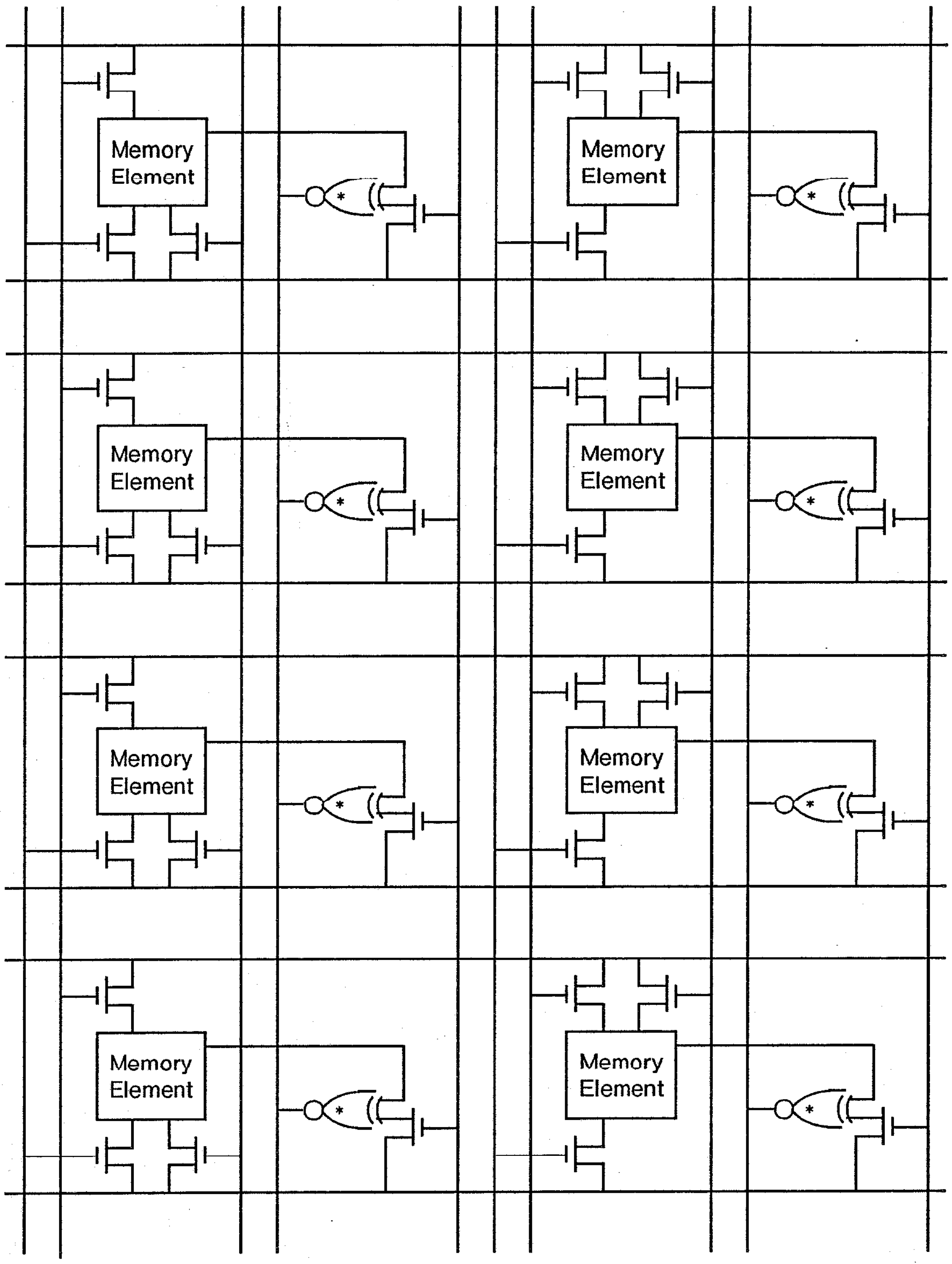


Figure 17b: Variable Value CAM Circuit Diagram

4. Fixed Value Double CAM

This datapath element is almost identical to the fixed value CAM. The only difference is that the fixed value double CAM divides the word to be loaded from the datapath into upper and lower halves, each of which is compared separately. The fixed value double CAM has only one load control line, but it has separate compare output lines for the upper and lower halves. A parameter in the runtime function specifies where the break occurs. In the Pascal system, this element is used to recognize stack bus message headers. Since the stack bus is half the width of a processor datapath, using double CAMs cuts the space requires in half.

Figure 18 shows two fixed value double CAMs. The comparison values are the same as those used in figure 16, except that here the comparison values are divided into upper and lower halves. The first CAM makes the division between bits 1 and 2; the second CAM makes the division between bits 0 and 1.

4.1 Runtime File

```
"DCAM1.ICL:         Runtime function definitions for double comparison
                    (separate high byte and low byte) content addressable
                    memories with fixed mask and fixed compare value       "

/*READ DCAM1D-CAM1D;*/       "also requires BINARY, MASK"


"Fixed value double CAM:  Description of Parameters

        Name:  name of block for block diagram
        U1:    microcode spec for loading word to be compared
        U2:    microcode spec for high byte comparison output
        U3:    microcode spec for low  byte comparison output
        Break: low order bit of the upper byte
        HVal:  mask constant indicating the high byte comparison value
        LVal:  mask constant indicating the low  byte comparison value
"

DEFINE DBL_CAM(NAME:SC  U1,U2,U3:USPEC_PRODUCER  BREAK:INT
                                                 HVAL,LVAL:MASK)=BLOCK_PRODUCER:
    //  [ NAME; U1; U2; U3; BREAK; HVAL; LVAL; ]
        BEGIN  VAR A, B = BLOCK; I = INT;
        DO
            IF BREAK > WIDTH-1 THEN WRITE('Byte boundary '); WRITE(BREAK);
                                    WRITE(' not less than datapath width');
                                    HELP;
            FI

            B := NEXT_GUY(12,12,5,5,6,8,1/6);
            A := NEXT_BLOCK;

            @(A).CALLS := { B          \AT 24#0;
                            DCAM1_COL \AT  0#0   \WITH_USPECS {U1;U2;U3};

                            FOR I FROM 0 TO BREAK-1; COLLECT
                                CASE TAIL(LVAL) OF
                                NONE:  NIL
                                ZERO:  DCAM1_ZERO \AT 0#I*HEIGHT
                                ONE:   DCAM1_ONE  \AT 0#I*HEIGHT
                            ENDCASE;

                        FOR I FROM BREAK TO WIDTH-1; COLLECT
                            CASE TAIL(HVAL) OF
                                NONE:  NIL
                                ZERO:  DCAM1_ZERO \AT 0#I*HEIGHT
                                ONE:   DCAM1_ONE  \AT 0#I*HEIGHT
                            ENDCASE
                      };

            @(A).VIEWS := { LAYOUT({ FOR I FROM 1 TO WIDTH-1;  WITH I<>BREAK;
                                     COLLECT  GW({ 15#Y4+I*HEIGHT;
                                                   .#Y2+6+I*HEIGHT })
                                 })
                      };
        GIVE A END
    \\
ENDDEFN

DEFINE DBL_CAM(NAME:SC  U1,U2,U3:USPEC_PRODUCER  BREAK:INT  HVAL,LVAL:MASK):
    DATAPATH ::= $> DBL_CAM(NAME,U1,U2,U3,BREAK,HVAL,LVAL);
ENDDEFN


"Multiple fixed value double CAM:  Description of Parameters

        Name:   name of block for block diagram
        U1:     microcode spec for loading word to be compared
        U2:     microcode spec for high byte comparison output
        U3:     microcode spec for low  byte comparison output
        Break:  low order bit of the upper byte
        HVals:  string of mask constants indicating high byte comp values
        LVals:  string of mask constants indicating low  byte comp values
"

DEFINE DBL_CAM(NAME:SC  U1,U2,U3:USPEC_PRODUCER  BREAK:INT  HVALS,LVALS:MASKS):
    BEGIN  VAR HVAL,LVAL = MASK;  I = INT;
        I := 1;
        WHILE  DEFINED(HVALS[I]) ! DEFINED(LVALS[I]);  DO
            DBL_CAM(NAME,U1,U2,U3,BREAK,DEF(HVALS[I]),DEF(LVALS[I])  );
            I := I+1;
        END
    END
ENDDEFN


"End of DCAM1.ICL"
```

4.2 Details File

```
"DCAM1D.ICL:        Details for double compare (separate high byte
                    and low byte) content addressable memories
                    with fixed mask and fixed compare value        "

VAR          DCAM1_NULL, DCAM1_NUM, DCAM1_ZERO = BLOCK;

VAR          DCAM1_ONE, DCAM1_CONN, DCAM1_COL  = BLOCK;
```

```
"PARAMETERS TO DISK_BLOCK ARE FILE NAME, POSITION IN FILE, UNIQUE ID"

DCAM1_NULL := DISK_BLOCK('DCAM1', 1, 4781);
DCAM1_NUM  := DISK_BLOCK('DCAM1', 2, 4782);
DCAM1_ZERO := DISK_BLOCK('DCAM1', 3, 4783);
DCAM1_ONE  := DISK_BLOCK('DCAM1', 4, 4784);
DCAM1_CONN := DISK_BLOCK('DCAM1', 5, 4785);
DCAM1_COL  := DISK_BLOCK('DCAM1', 6, 4786);

"End of DCAM1D.ICL"
```

4.3 Source File

```
"DCAM1S.ICL:        Definitions fordouble compare (separate high byte
                    and low byte) content addressable memories
                    with fixed mask and fixed compare value      "

/*READ DCAM1D-CAM1D;*/


VAR         DCAM1_VNULL, DCAM1_VNUM, DCAM1_VZERO, DCAM1_VONE,
                                     DCAM1_VCONN, DCAM1_VCOL  = VBLOCK;


DCAM1_VNULL       := NIL;               "Contains bus wires & control lines"

DCAM1_VNULL.VIEWS :=  { LAYOUT ({

        BW ({ 0#0\Y1; 24#.        });          "upper data bus wire"
        BW ({ 0#0\Y2; 24#.        });          "lower data bus wire"

        RW ({ 4#0\Y4; .#2.5\Y2; .-2#.+2;
              .#-4.5\Y1; .+2#.+2; .#0\Y3 });   "data load line"
        RW ({ 20#0\Y4; .#2.5\Y2; 22#.+2;
              .#-4.5\Y1; 20#.+2; .#0\Y3  });   "high byte compare return line"

        GW ({ 15#6\Y2; 18.5#.;
              .#-6\Y1; 15#.; .#0\Y3      })    "compare line"
    }) };


DCAM1_VNUM        := NIL;               "Defines basic comparison cell"

DCAM1_VNUM.CALLS  := { CAM1_NUM };


DCAM1_VZERO       := NIL;

DCAM1_VZERO.CALLS := { CAM1_ZERO };


DCAM1_VONE        := NTI ;

DCAM1_VONE.CALLS  := { CAM1_ONE };


DCAM1_VCONN       := NIL;     "Defines connections at top and bottom of column"
```

```
DCAM1_VCONN.CALLS  := { GRCBL \AT 7#1\Y6;      "load line in contact"
                        GRCBR \AT 16#-1\Y5;    "return line top contact"
                        GRCBU \AT 20#1\Y6      "high byte equal out contact"
                      };

DCAM1_VCONN.VIEWS  := { LAYOUT ({
        RW ({ 20#0\Y4; .#5\Y6         });     "high byte equal out connect"
        RW ({  4#3\Y6; .#0\Y4         });     "data load connect"
        RW ({ 20#0\VY TIP_TOP; .#0\Y5 });     "top return line connect"

        GW ({ 15#0\VY TIP_TOP; .#0\Y5 });     "top return line connect"
        GW ({ 14#0\Y6; .#6\Y2         })      "low byte compare line connect"
    }) };

DCAM1_VCONN.INTERFACE := { [ FROM: 8#0\Y6 TYPE:1 BUFFER:1 EDGE:3 UCODE:1 ];
                           [ FROM:14#0\Y6 TYPE:1 BUFFER:3 EDGE:3 UCODE:3 ];
                           [ FROM:20#0\Y6 TYPE:1 BUFFER:3 EDGE:3 UCODE:2 ] };


DCAM1_VCOL         := NIL;

DCAM1_VCOL.CALLS   := {  DCAM1_CONN  PASSING_USPECS {1;2;3};
                         DCAM1_NULL \AT EACH_BIT_POSITION  };


SCRATCH_FILE(DCAM1_NULL.DETAILS.FILE_NAME);


DUMP_SET({  [ B: DCAM1_NULL   V: DCAM1_VNULL ];
            [ B: DCAM1_NUM    V: DCAM1_VNUM  ];
            [ B: DCAM1_ZERO   V: DCAM1_VZERO ];
            [ B: DCAM1_ONE    V: DCAM1_VONE  ];
            [ B: DCAM1_CONN   V: DCAM1_VCONN ];
            [ B: DCAM1_COL    V: DCAM1_VCOL  ]
        });


"End of DCAM1S.ICL"
```

4.4 Test File

```
"DCAM1T.ICL:          Tests double comparison (separate high
                      and low bute) content addressable memories
                      with fixed mask and fixed compare value     "


VAR  A = BLOCK;

PRECHARGE := FALSE;
DATA_WIDTH(4);

DBL_CAM( 'Cam', UNUSED, UNUSED, UNUSED, 2,  '00', '11' );
DBL_CAM( 'Cam', UNUSED, UNUSED, UNUSED, 1, '1XX',  '0' );

A := DATAPATH;


"End of DCAM1T.ICL"
```

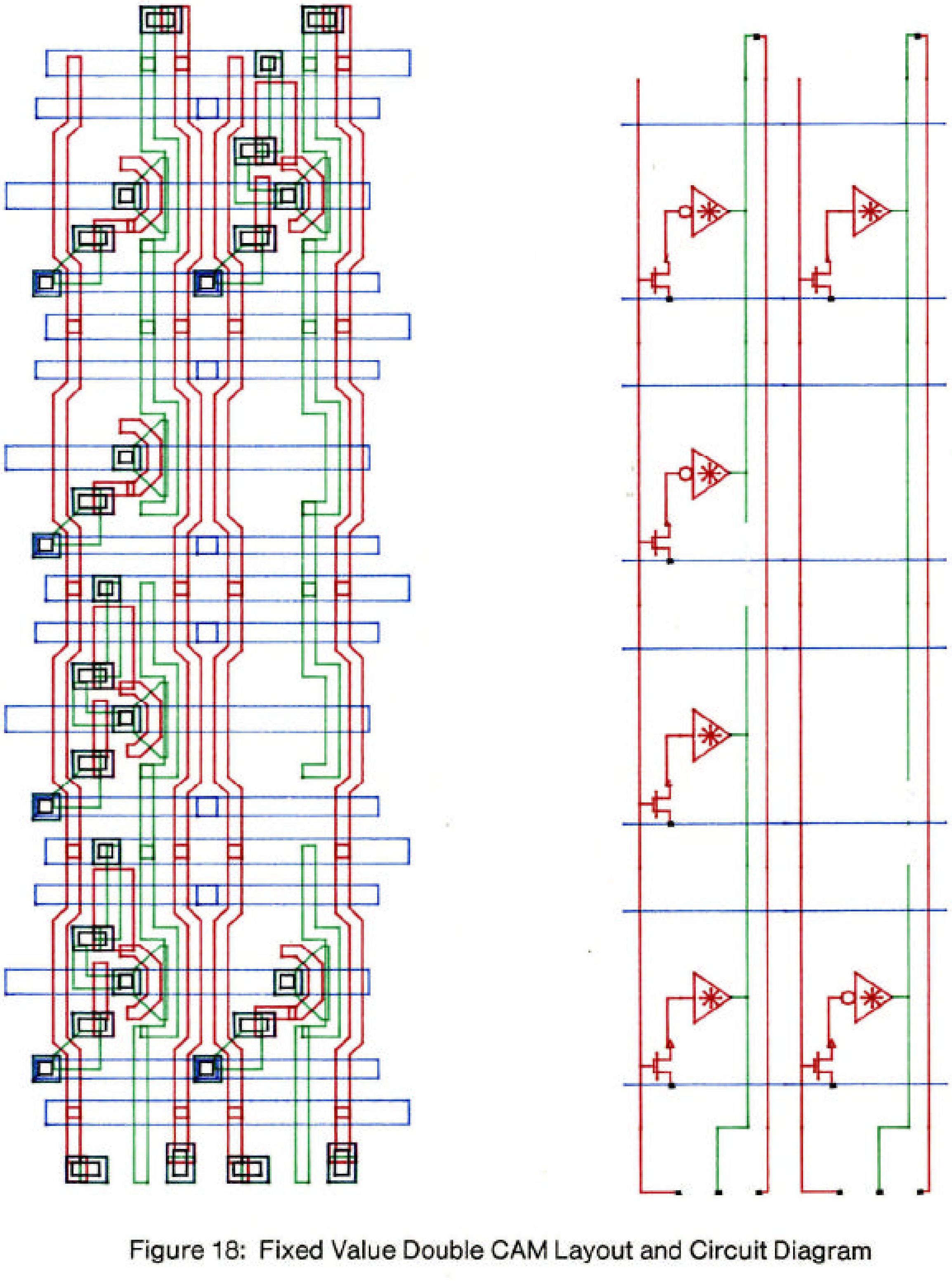


Figure 18: Fixed Value Double CAM Layout and Circuit Diagram

5. Variable Value Double CAM

This datapath element is almost identical to the variable value CAM. The only difference is that the variable value double CAM divides the word to be compared and the word to be loaded from the datapath into upper and lower halves, just as the fixed value double CAM does. This datapath element does not occur in the Pascal system. However, it could be substituted for the fixed value double CAM if the stack bus message headers were not specified prior to chip fabrication.

Figure 19 shows two variable value double CAMs. The first compares against all four bits. The second ignores bits 1 and 3. The comparison value is stored in a standard memory register. The first CAM's memory register can be written from the upper bus and read onto the lower bus. The second CAM's memory register can be read from the upper bus and written onto the lower bus. Both memory registers have refresh control lines (not shown in the circuit diagram). Normally these are driven by the Ø1 clock.

5.1 Runtime File

```
"DCAM2.ICL:        Runtime function definitions for double compare
                   (separate high byte and low byte) content addressable
                   memories with fixed mask and variable compare value    "

/*READ DCAM2D-CAM2D-L01D;*/      "also requires BINARY.ICL"


"Variable value double CAM with no Masking:  Description of Parameters"

"        Name:  name of block for block diagram"
"        U1:    microcode spec for loading word to be compared"
"        U2:    microcode spec for high byte comparison output"
"        U3:    microcode spec for low  byte comparison output"
"        MU:    microcode specs for comparison register"
"        Break: low order bit of high byte"


DEFINE DBL_CAM(NAME:SC  U1,U2,U3:USPEC_PRODUCER  MU:MEMORY_USPEC
                                                 BREAK:INT)=BLOCK_PRODUCER:
    //  [ NAME; U1; U2; U3; MU; BREAK; ]
        BEGIN  VAR A, B = BLOCK; I = INT; MR = MEMORY_RETURN; R = REAL;
        DO
            IF BREAK > WIDTH-1 THEN WRITE('Byte boundary '); WRITE(BREAK);
                                    WRITE(' not less than datapath width');
                                    HELP;
            FI

            MR  := MEMORY_REGISTER(MU,[A:0 B:0],[A:1 B:0],[A:0 B:0],[A:0 B:0]);
            B   := NEXT_GUY(14,14,5,5,6,8,1/8+1/4+MR.POWER);
            A   := NEXT_BLOCK;
            R   := MR.WIDTH;

            @(A).CALLS := { B          \AT 35+R#0;
                            DCAM2_ALL  \AT R#0 \WITH_USPECS {U1;U2;U3}
```

```
                    };

            @(A).VIEWS := { LAYOUT({ FOR I FROM 1 TO WIDTH-1; WITH I<>BREAK;
                                     COLLECT  GW({ 10+R#Y4+I*HEIGHT;
                                                   .#Y2+3+I*HEIGHT })
                                 });
                            BLOCK(NAME,35+R,DEFINED(MU.RU),DEFINED(MU.WU),
                                      TRUE,DEFINED(MU.RL)   )       };
        GIVE A END
    \\
ENDDEFN


DEFINE DBL_CAM(NAME:SC U1,U2,U3:USPEC_PRODUCER MU:MEMORY_USPEC BREAK:INT):
    DATAPATH ::= $> DBL_CAM(NAME,U1,U2,U3,MU,BREAK);
ENDDEFN


"Variable value double CAM with Fixed Mask:  Description of Parameters"

"         Name:  Name of block for block diagram"
"         U1:    Microcode spec for loading word to be compared"
"         U2:    Microcode spec for high byte comparison output"
"         U3:    Microcode spec for low  byte comparison output"
"         MU:    Microcode specs for comparison register"
"         Break: Low order bit of high byte"
"         HVal:  Binary constant indicating which bits in high byte are compared.
                 Bits marked zero are ignored in the comparison."
"         LVal:  Binary constant indicating which bits in low  byte are compared.
                 Bits marked zero are ignored in the comparison."

DEFINE DBL_CAM(NAME:SC  U1,U2,U3:USPEC_PRODUCER  MU:MEMORY_USPEC
                                 BREAK:INT  HVAL,LVAL:BINARY) = BLOCK_PRODUCER:
    //  [ NAME; U1; U2; U3; MU; BREAK; HVAL; LVAL; ]
        BEGIN  VAR A, B = BLOCK; I = INT; MR = MEMORY_RETURN; R = REAL;
        DO
            IF BREAK > WIDTH-1 THEN WRITE('Byte boundary '); WRITE(BREAK);
                                    WRITE(' not less than datapath width');
                                    HELP;
            FI

            MR := MEMORY_REGISTER(MU,[A:0 B:0],[A:1 B:0],[A:0 B:0],[A:0 B:0]);
            B  := NEXT_GUY(14,14,5,5,6,8,1/8+1/4);
            A  := NEXT_BLOCK;
            R  := MR.WIDTH;


            @(A).CALLS := { B         \AT 35+R#0;
                            DCAM2_COL \AT R#0 \WITH_USPECS {U1;U2;U3};

                            FOR I FROM 0 TO BREAK-1; COLLECT
                                IF TAIL(LVAL) THEN DCAM2_CELL \AT R#I*HEIGHT
                                              ELSE NIL
                                FI;

                            FOR I FROM BREAK TO WIDTH-1; COLLECT
                                IF TAIL(HVAL) THEN DCAM2_CELL \AT R#I*HEIGHT
                                              ELSE NIL
                                FI
                        };

            @(A).VIEWS := { LAYOUT({ FOR I FROM 1 TO WIDTH-1;  WITH I<>BREAK;
                                             COLLECT  GW({ 10+R#Y4+I*HEIGHT;
                                                              .#Y2+3+I*HEIGHT })
                                       });
                               BLOCK(NAME,35+R,DEFINED(MU.RU),DEFINED(MU.WU),
                                               TRUE,DEFINED(MU.RL)    )        };
        GIVE A END
    \\
ENDDEFN

DEFINE DBL_CAM(NAME:SC  U1.U2.U3:USPEC_PRODUCER  MU:MEMORY_USPEC  BREAK:INT
                                                             HVAL,LVAL:BINARY):
    DATAPATH ::= $> DBL_CAM(NAME,U1,U2,U3,MU,BREAK,HVAL,LVAL);
ENDDEFN


"End of DCAM2.ICL"
```

5.2 Details File

```
"DCAM2D.ICL:        Details for double compare (separate high byte
                    and low byte) content addressable memories
                    with fixed mask and variable compare value"

VAR         DCAM2_NULL, DCAM2_CELL, DCAM2_CONN, DCAM2_COL, DCAM2_ALL = BLOCK;

"PARAMETERS TO DISK_BLOCK ARE FILE NAME, POSITION IN FILE, UNIQUE ID"

DCAM2_NULL := DISK_BLOCK('DCAM2', 1, 8281);
DCAM2_CELL := DISK_BLOCK('DCAM2', 2, 8282);
DCAM2_CONN := DISK_BLOCK('DCAM2', 3, 8283);
DCAM2_COL  := DISK_BLOCK('DCAM2', 4, 8284);
DCAM2_ALL  := DISK_BLOCK('DCAM2', 5, 8285);

"End of DCAM2D.ICL"
```

5.3 Source File

```
"DCAM2S.ICL:        Definitions for double compare (separate high byte
                    and low byte) content addressable memories
                    with fixed mask and variable compare value   "

/*READ DCAM2D-CAM2D;*/


VAR   DCAM2_VNULL, DCAM2_VCELL, DCAM2_VCONN, DCAM2_VCOL, DCAM2_VALL = VBLOCK;


DCAM2_VNULL       := NIL:                "Contains bus wires & control lines"


DCAM2_VNULL.VIEWS :=  { LAYOUT ({

        BW ({ 0#0\Y1; 35#.        });   "upper data bus wire"
        BW ({ 0#0\Y2; 35#.        });   "lower data bus wire"

        RW ({ 31#0\Y4; .#2.5\Y2;
              33#.+2; .#-4.5\Y1;
              31#.+2; .#0\Y3      });   "data load control line"
        RW ({ 4#0\Y4; .#2.5\Y2;
            2#.+2;  .#-4.5\Y1;
            4#.+2;  .#0\Y3      });   "high byte equal return line"

        GW ({ 9.5#4\Y2;
            5.5#.+4.5; .#-8\Y1;
            10#.+4.5: .#0\Y3    })    "equal line"
    }) };


DCAM2_VCELL       := NIL;              "Defines basic comparison cell"

DCAM2_VCELL.CALLS := { CAM2_CELL \AT 4#0 };


DCAM2_VCONN       := NIL;      "Defines connections at bottom and top of column"

DCAM2_VCONN.CALLS :=
                     {   GRCBU \AT 29#1\Y6;   "load line contact"
                         GRCBL \AT 8#-1\Y5;   "high byte return line contact"
                         GRCBU \AT  4#1\Y6    "high byte equal out contact"
                     };

DCAM2_VCONN.VIEWS :=
    { LAYOUT ({
        RW ({ 31#0\Y4; .#3\Y6  });         "load line input connect"
        RW ({  4#0\Y4; .#3\Y6  });         "high byte equal out connect"
        RW ({ 4#0\VY TIP_TOP; .#0\Y5 });   "top return line connect"

        GW ({ 10#0\VY TIP_TOP; .#0\Y5 });  "top return line connect"
        GW ({ 10#0\Y6; .#3\Y2  })          "low byte equal out connect"
    }) };

DCAM2_VCONN.INTERFACE := { [ FROM:29#0\Y6 TYPE:1 BUFFER:1 EDGE:3 UCODE:1 ];
                           [ FROM:10#0\Y6 TYPE:1 BUFFER:3 EDGE:3 UCODE:3 ];
                           [ FROM: 4#0\Y6 TYPE:1 BUFFER:3 EDGE:3 UCODE:2 ] };


DCAM2_VCOL       := NIL;          "Defines a CAM without any comparison cells"

DCAM2_VCOL.CALLS := { DCAM2_NULL    \AT 0#0     \AT EACH_BIT_POSITION;
                      DCAM2_CONN    \AT 0#0      PASSING_USPECS {1;2;3}  };


DCAM2_VALL       := NIL;          "Defines a CAM with comp cells for each bit"

DCAM2_VALL.CALLS := { DCAM2_COL   PASSING_USPECS {1;2;3};
                      DCAM2_CELL \AT  EACH_BIT_POSITION   };


SCRATCH_FILE(DCAM2_CELL.DETAILS.FILE_NAME);

DUMP_SET({  [ B: DCAM2_NULL   V: DCAM2_VNULL ];
            [ B: DCAM2_CELL   V: DCAM2_VCELL ];
            [ B: DCAM2_CONN   V: DCAM2_VCONN ];
            [ B: DCAM2_COL    V: DCAM2_VCOL  ];
            [ B: DCAM2_ALL    V: DCAM2_VALL  ]
        });

"End of DCAM2S.ICL"
```

5.4 Test File

```
"DCAM2T.ICL:          Tests double compare (separate high byte
                      and low byte) content addressable memories
                      with fixed mask and variable compare value  "


VAR  A = BLOCK;

PRECHARGE :- FALSE;
DATA_WIDTH(4);


DBL_CAM( 'Cam', UNUSED, UNUSED, UNUSED, [RU:UNUSED WL:UNUSED], 2              );
DBL_CAM( 'Cam', UNUSED, UNUSED, UNUSED, [RL:UNUSED WL:UNUSED], 1, '100', '1' );

A := DATAPATH;


"End of DCAM2T.ICL"
```

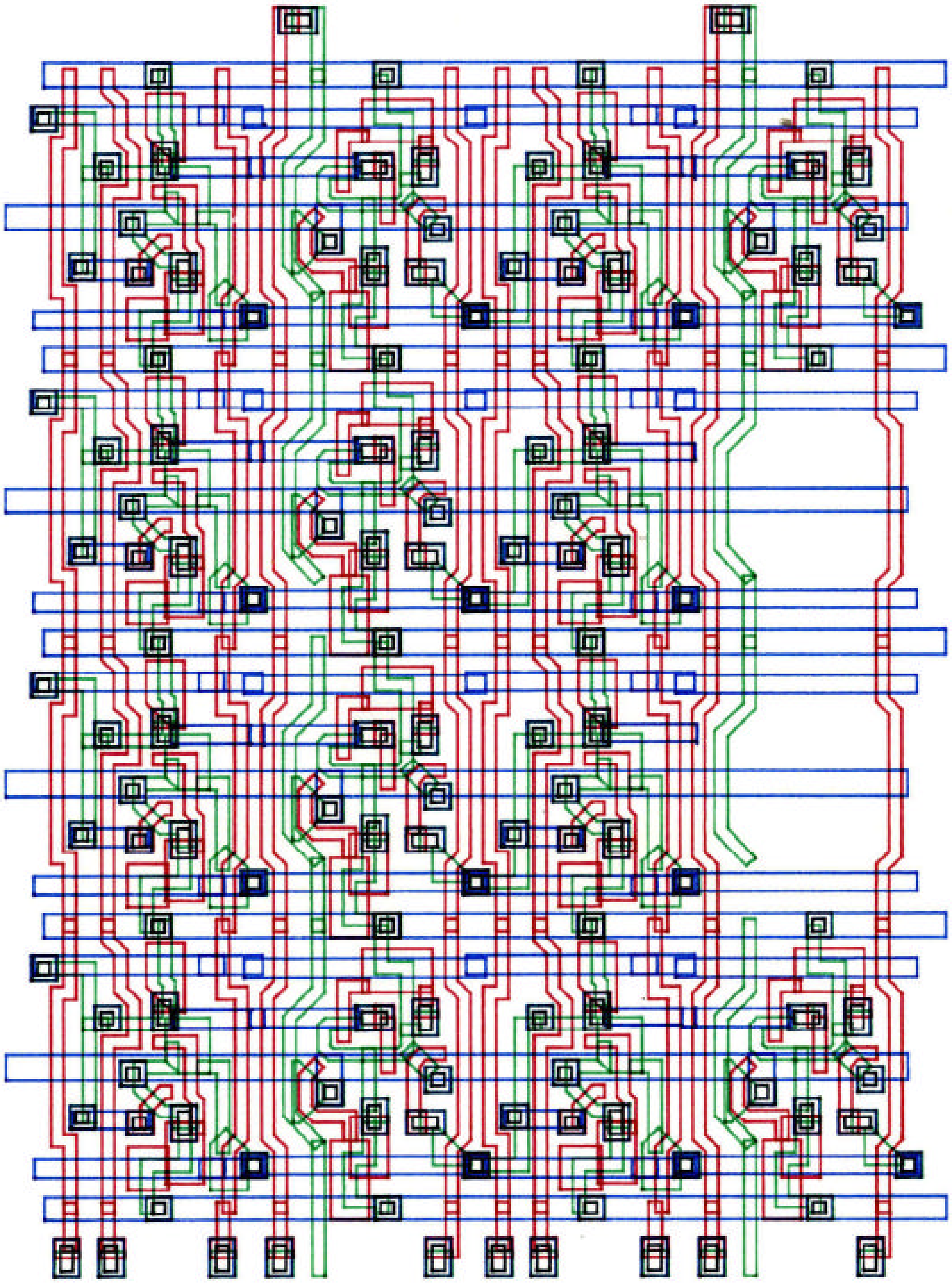


Figure 19a: Variable Value Double CAM Layout

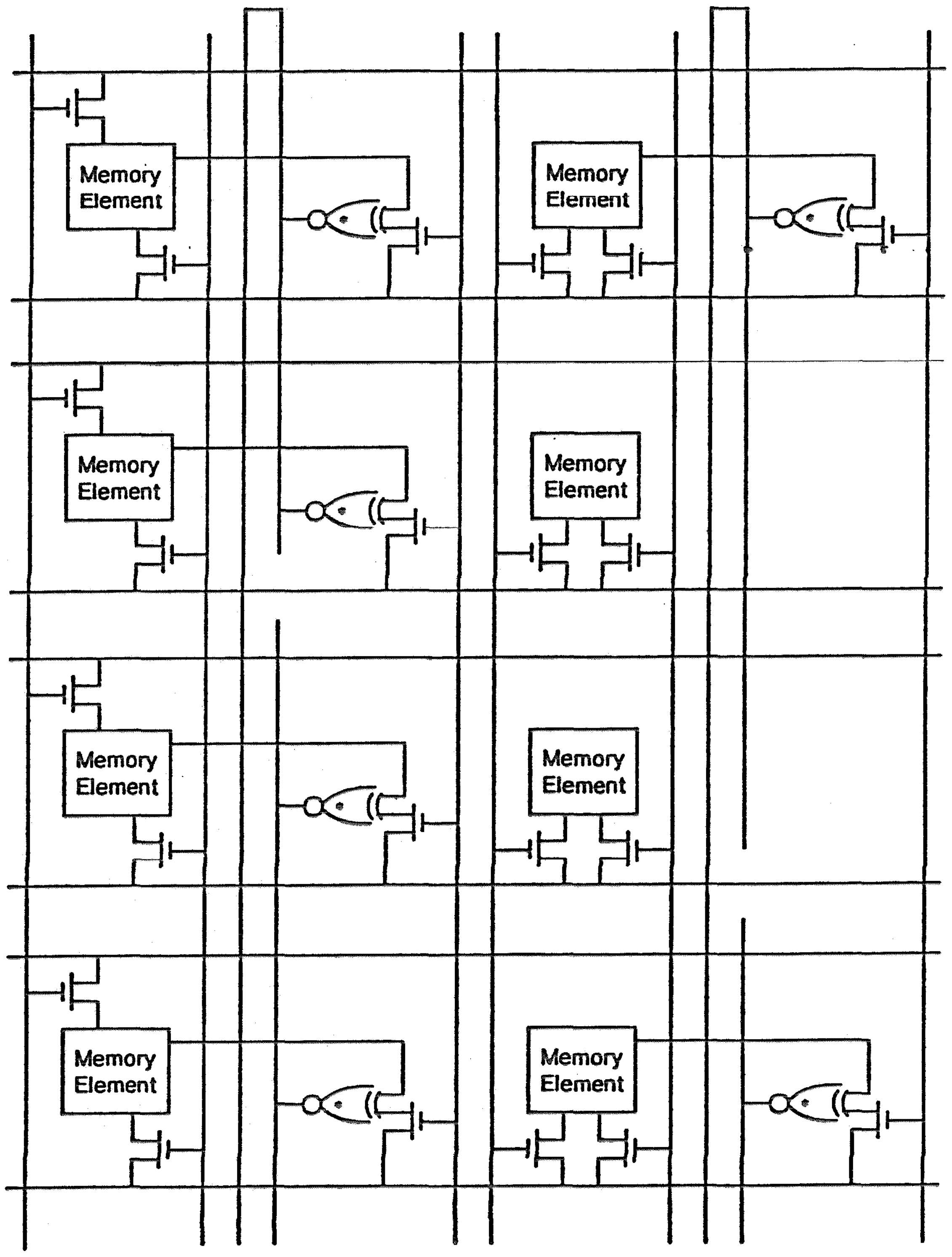


Figure 19b: Variable Value Double CAM Circuit Diagram

Appendix B: EM-1 Instruction and Frequency Tables

The sections in this appendix define Tannenbaum's EM-1 instruction set and give information about static instruction frequencies. For each instruction, section 1 lists the mnemonic, operand type, instruction group, and a brief description. Sections 2 through 4 leave out the instruction description information but include the execution processor, the special case code, the number of words popped and pushed, and static instruction frequency data. Tannenbaum did not indicate the source of the frequency data. Possibly it was collected from a Pascal compiler. In section 2, the list is sorted by execution processor, in section 3 it is sorted by frequency, and in section 4 it is sorted by instruction mnemonic. Section 5 lists all mnemonic and operand combinations which occur more than 40 times in Tannenbaum's test code. The list includes 492 elements, and is sorted alphabetically by mnemonic.

Instr Group	Meaning
Load	Load instructions
Store	Store instructions
Int	Integer arithmetic instructions
Double	Double precision integer arithmetic instructions
Real	Flooating point arithmetic instructions
Convert	Conversion instructions
Boolean	Boolean operation instructions
Set	Set manipulation instructions
Array	Array access instructions
Compare	Comparison instructions
Branch	Branch instructions
Call	Procedure call instructions
Inc/Dec	Increment/decrement/zero instructions
Misc	Miscellaneous instructions
Monitor	Monitor instructions

Table 11: Instruction Groups

Operand Type	Meaning
(blank)	Zero operand instruction
x	Byte count, must be even
y	Byte count, must be 1 or even
b	Byte count, must be greater than or equal to zero
n	Nonnegative integer
m	Integer operand

Table 12: Operand Types

Exec Unit	Meaning
Int	Integer processor
Real	Floating point processor
Stack	Stack manager
Mem	Memory manager
Instr	Instruction unit

Table 13: Instruction Execution Processors

Code Field	Meaning
(blank)	No special case
m	Instruction generated by the assembler (machine only instruction)
*	Instruction is preceeded by a load stack register (lor 3) instruction
!	Procedure call or return instruction, special execution sequence
x	Instruction is not issued over stack bus

Table 14: Special Case Codes

Push/Pop Char	Meaning
number	Push/pop that number of words
x or y	Push/pop instruction operand/2 words
a	Push/pop address: one word for non virtual addressing, two words for virtual addressing
↑	Push/pop number of words indicated by top of stack
#	Push/pop array element
???	Push/pop an unknown number of words

Table 15: Stack Push/Pop Codes

1. Instruction Table, Sorted by Instruction Group

This section lists the EM-1 instruction set and provides a brief description of each instruction. The list is organized by instruction groups, which are described in t able 11.

Opcode, Operand		Instr Group	Instruction Description
loc	m	load	Load constant (i.e. push it onto the stack)
lnc	n	load	Load negative constant
lol	x	load	Load local word x
loe	x	load	Load external word x
lop	x	load	Load parameter (address is at xth local)
lof	x	load	Load offsetted (top of stack + x yields address)
lal	x	load	Load address of local
lae	x	load	Load address of external
lex	n	load	Load lexical (address of lb, n static levels back)
loi	y	load	Load indirect y bytes (address is popped from the stack)
lib		load	Load indirect 1 byte (loaded into 1 word on stack)
los		load	Load indirect (first pop byte count; must be 1 or even)
ldl	x	load	Load double local (two consecutive locals are stacked)
lde	x	load	Load double external
ldf	x	load	Load double offsetted (top of stack + x yields address)
stl	x	store	Store local
ste	x	store	Store external
stp	x	store	Store parameter
stf	x	store	Store offsetted
sti	y	store	Store indirect y bytes (pop address, then data)
sib		store	Store indirect 1 byte (taken from 1 word on stack)
sts		store	Store indirect (pop byte count, then address, then data)
sdl	x	store	Store double local
sde	x	store	Store double external
sdf	x	store	Store double offsetted
add		integer	Integer add
sub		integer	Integer subtract
mul		integer	Integer multiply
div		integer	Integer divide
mod		integer	Modulo (remainder)
neg		integer	Negate (two's complement)
shl		integer	Shift left tos-1 by tos bits
shr		integer	Shift right tos-1 by tos bits
rol		integer	Rotate left tos-1 by tos bits
ror		integer	Rotate right tos-1 by tos bits
inc		integer	Increment top of stack by 1
dec		integer	Decrement top of stack by 1
exg		integer	Exchange top two words
adi	b	integer	Add the constant b to top of stack; do not check overflow
ads		integer	Same as add, but do not check for overflow
dad		double	Double add
dsb		double	Double subtract
dmu		double	Double multiply

Appendix B: EM-1 Instruction and Frequency Tables

Section 1: Instruction Table, Sorted by Instruction Group

Opcode,	Operand	Instr Group	Instruction Description
ddv		double	Double divide
dmd		double	Double modulo
fad		real	Floating add
fsb		real	Floating subtract
fmu		real	Floating multiply
fdv		real	Floating divide
cid		convert	Convert integer to double
cdi		convert	Convert double to integer
cif		convert	Convert integer to floating
cfi		convert	Convert floating to integer
cdf		convert	Convert double to floating
cfd		convert	Convert floating to double
and	x	boolean	Boolean and on two groups of x bytes
and	2	boolean	Boolean and on two groups of 2 bytes
ans		boolean	Boolean and; first pop number of bytes from stack
ior	x	boolean	Boolean inclusive or on two groups of x bytes
ior	2	boolean	Boolean inclusive or on two groups of 2 bytes
ios		boolean	Boolean inclusive or; first pop number of bytes from stack
xor	x	boolean	Boolean exclusive or on two groups of x bytes
xos		boolean	Boolean exclusive or; first pop number of bytes from stack
com	x	boolean	Complement (one's complement of top x bytes)
cos		boolean	Complement; first pop number of bytes from stack
not		boolean	Convert top of stack from true to false or vice versa
inn	x	set	Bit test on x byte set (bit number on top of stack)
ins		set	Bit test; first pop set size, then bit number
set	x	set	Create singleton x word set with bit number tos on
ses		set	Create singleton set; first pop set size, then bit number
lar	x	array	Load array element with descriptor x, pop index, array addr
las		array	Load array element; first pop pointer to descriptor
sar	x	array	Store array element; pop index, array address, array element
sas		array	Store array element; first pop pointer to descriptor
aar	x	array	Stack address of array element; pop index, array address
aas		array	Stack array address; first pop pointer to descriptor
cmi		compare	Compare two integers. Push -1,0,1 for <, = ,>
cmd		compare	Compare two double integers
cmf		compare	Compare two reals
cmu	x	compare	Compare two blocks of x bytes each
cms		compare	Compare two blocks of bytes; pop byte count
tlt		compare	True if less (based on previous compare)
tle		compare	True if less or equal
teq		compare	True if equal
tne		compare	True if not equal
tge		compare	True if greater or equal
tgt		compare	True if greater
brf	n	branch	Branch foreward unconditionally n bytes
brb	n	branch	Branch backward unconditionally n bytes
blt	n	branch	Forward branch less (pop two words, branch if top > second)
ble	n	branch	Forward branch less or equal

Appendix B: EM-1 Instruction and Frequency Tables

Section 1: Instruction Table, Sorted by Instruction Group

Opcode, Operand		Instr Group	Instruction Description
beq	n	branch	Forward branch equal
bne	n	branch	Forward branch not equal
bge	n	branch	Forward branch greater or equal
bgt	n	branch	Forward branch greater
zlt	n	branch	Forward branch less than zero (pop 1 word, branch negative)
zle	n	branch	Forward branch less or equal to zero
zeq	n	branch	Forward branch equal zero
zne	n	branch	Forward branch not zero
zge	n	branch	Forward branch greater or equal to zero
zgt	n	branch	Forward branch greater than zero
mrk	n	call	Mark stack (n = 1 + change in static depth of nesting)
mrx	n	call	Mark stack; set alternate context
mrs		call	Mark stack; first pop static link from stack
mxs		call	Mark stack; pop static link and set alternate context
cal	n	call	Call procedure with descriptor n
cas		call	Call indirect (first pop procedure number from stack)
ret	n	call	Return (function result consists of top x bytes)
res		call	Return; first pop number of bytes to return
inl	x	inc/dec	Increment local
ine	x	inc/dec	Increment external
del	x	inc/dec	Decrement local
dee	x	inc/dec	Decrement external
zrl	x	inc/dec	Zero local
zre	x	inc/dec	Zero external
beg	x	misc	Begin procedure (reserve x bytes for locals)
bes		misc	Begin procedure; pop number of bytes to reserve for locals
rck	x	misc	Range check (trap if top of stack out of range)
nop		misc	No operation
blm	x	misc	Block move x bytes; pop source address, then destination
bls		misc	Block move; first pop x, then addresses
lin	n	misc	Line number (set external 0 to n)
dup	x	misc	Duplicate top x words on stack
dup	2	misc	Duplicate top 2 words on stack
dus		misc	Duplicate; first pop number of words to duplicate
cse	x	misc	Case jump; x is external offset of jump table
lor	n	misc	Load EM-1 machine register onto stack
lor	1	misc	Load procedure descriptors register
lor	2	misc	Load local base register
lor	3	misc	Load stack pointer register
lor	4	misc	Load heap pointer register
str	n	misc	Store EM-1 machine register from stack
str	1	misc	Store procedure descriptor register
str	2	misc	Store local base register
str	3	misc	Store stack pointer register
str	4	misc	Store heap pointer register
hlt		misc	Halt the machine
mon		monitor	Monitor call
stu		monitor	Start user job

2. Mnemonic Table, Sorted by Execution Unit

This section lists information about the EM-1 mnemonics. The list is sorted by the execution units, described in table 13.

Opcode, Operand		Instr. Group	Exec. Unit	Code	Words Popped	Words Pushed	Static Freq.	Percent
loc	m	load	instr		0	1	27604	14.79 %
lnc	n	load	instr	m	0	1	1136	0.61 %
brf	n	branch	instr	x	0	0	3067	1.64 %
brb	n	branch	instr	x	0	0	2688	1.44 %
blt	n	branch	instr		2	0	1016	0.54 %
ble	n	branch	instr		2	0	305	0.16 %
beq	n	branch	instr		2	0	1060	0.57 %
bne	n	branch	instr		2	0	1681	0.90 %
bge	n	branch	instr		2	0	293	0.16 %
bgt	n	branch	instr		2	0	598	0.32 %
zlt	n	branch	instr		1	0	47	0.03 %
zle	n	branch	instr		1	0	217	0.12 %
zeq	n	branch	instr		1	0	2094	1.12 %
zne	n	branch	instr		1	0	1139	0.61 %
zge	n	branch	instr		1	0	155	0.08 %
zgt	n	branch	instr		1	0	31	0.02 %
nop		misc	instr	x	0	0	0	0.00 %
lin	n	misc	instr		0	0	1763	0.94 %
cse	x	misc	instr		1	0	297	0.16 %
lor	n	misc	instr		0	1	0	0.00 %
str	n	misc	instr		1	0	0	0.00 %
str	1	misc	instr	m	1	0	0	0.00 %
hlt		misc	instr		0	0	0	0.00 %
mon		monitor	instr		0	0	5	0.00 %
stu		monitor	instr		8	0	0	0.00 %
add		integer	int		2	1	1469	0.79 %
sub		integer	int		2	1	878	0.47 %
mul		integer	int		2	1	565	0.30 %
div		integer	int		2	1	383	0.21 %
mod		integer	int		2	1	343	0.18 %
neg		integer	int		1	1	108	0.06 %
shl		integer	int		2	1	139	0.07 %
shr		integer	int		2	1	0	0.00 %
rol		integer	int		2	1	0	0.00 %
ror		integer	int		2	1	0	0.00 %
inc		integer	int		1	1	797	0.43 %
dec		integer	int		1	1	478	0.26 %
exg		integer	int		2	2	0	0.00 %
adi	b	integer	int		1	1	960	0.51 %
ads		integer	int		2	1	0	0.00 %
dad		double	int		4	2	0	0.00 %
dsb		double	int		4	2	0	0.00 %
dmu		double	int		4	2	0	0.00 %

Appendix B: EM-1 Instruction and Frequency Tables

Section 2: Mnemonic Table, Sorted by Execution Unit

Opcode, Operand		Instr. Group	Exec. Unit	Code	Words Popped	Words Pushed	Static Freq.	Percent
ddv		double	int		4	2	0	0.00 %
dmd		double	int		4	2	0	0.00 %
cid		convert	int		1	2	0	0.00 %
cdi		convert	int		2	1	0	0.00 %
and	2	boolean	int	m	2	1	1035	0.55 %
ior	2	boolean	int	m	2	1	726	0.39 %
not		boolean	int		1	1	0	0.00 %
cmi		compare	int		2	1	1911	1.02 %
cmd		compare	int		4	1	0	0.00 %
tlt		compare	int		1	1	292	0.16 %
tle		compare	int		1	1	172	0.09 %
teq		compare	int		1	1	1184	0.63 %
tne		compare	int		1	1	591	0.32 %
tge		compare	int		1	1	134	0.07 %
tgt		compare	int		1	1	290	0.16 %
dup	2	misc	int	m	1	2	845	0.45 %
lol	x	load	mem		0	1	17140	9.18 %
loe	x	load	mem		0	1	9490	5.08 %
lop	x	load	mem		0	1	676	0.36 %
lof	x	load	mem		1	1	3472	1.86 %
lal	x	load	mem		0	a	1805	0.97 %
lae	x	load	mem		0	a	16242	8.70 %
lex	n	load	mem		0	a	1885	1.01 %
loi	y	load	mem		a	y	1561	0.84 %
lib		load	mem	m	a	1	0	0.00 %
los		load	mem		1 + a	↑	0	0.00 %
ldl	x	load	mem		0	2	1576	0.84 %
lde	x	load	mem		0	2	1196	0.64 %
ldf	x	load	mem		1	2	73	0.04 %
stl	x	store	mem		1	0	6587	3.53 %
ste	x	store	mem		1	0	3588	1.92 %
stp	x	store	mem		1	0	731	0.39 %
stf	x	store	mem		2	0	1788	0.96 %
sti	y	store	mem	*	a + y	0	746	0.40 %
sib		store	mem	m	a + 1	0	0	0.00 %
sts		store	mem	*	1 + a + ↑	0	0	0.00 %
sdl	x	store	mem		2	0	377	0.20 %
sde	x	store	mem		2	0	390	0.21 %
sdf	x	store	mem		a + 1	0	60	0.03 %
lar	x	array	mem		1 + a	#	2111	1.13 %
las		array	mem		2 + a	#	0	0.00 %
sar	x	array	mem	*	1 + a + #	0	1245	0.67 %
sas		array	mem	*	2 + a + #	0	0	0.00 %
aar	x	array	mem		1 + a	a	1753	0.94 %
aas		array	mem		2 + a	a	0	0.00 %
mrk	n	call	mem		0	3	19147	10.26 %
mrx	n	call	mem	m	0	3	0	0.00 %
mrs		call	mem		1	3	4	0.00 %

Appendix B: EM-1 Instruction and Frequency Tables

Section 2: Mnemonic Table, Sorted by Execution Unit

Opcode, Operand		Instr. Group	Exec. Unit	Code	Words Popped	Words Pushed	Static Freq.	Percent
mxs		call	mem	m	1	3	0	0.00 %
inl	x	inc/dec	mem		0	0	1403	0.75 %
ine	x	inc/dec	mem		0	0	1285	0.69 %
del	x	inc/dec	mem		0	0	126	0.07 %
dee	x	inc/dec	mem		0	0	71	0.04 %
zrl	x	inc/dec	mem		0	0	784	0.42 %
zre	x	inc/dec	mem		0	0	760	0.41 %
rck	x	misc	mem		1	1	987	0.53 %
blm	x	misc	mem		a + a	0	482	0.26 %
bls		misc	mem		1 + a + a	0	0	0.00 %
lor	1	misc	mem	m	0	1	0	0.00 %
lor	2	misc	mem	m	0	1	0	0.00 %
lor	4	misc	mem	m	0	1	0	0.00 %
str	2	misc	mem	m	1	0	0	0.00 %
str	4	misc	mem	m	1	0	0	0.00 %
fad		real	real		4	2	355	0.19 %
fsb		real	real		4	2	301	0.16 %
fmu		real	real		4	2	482	0.26 %
fdv		real	real		4	2	289	0.15 %
cif		convert	real		1	2	1227	0.66 %
cfi		convert	real		2	1	303	0.16 %
cdf		convert	real		2	2	0	0.00 %
cfd		convert	real		2	2	0	0.00 %
cmf		compare	real		4	1	338	0.18 %
and	x	boolean	stack		x + x	x	55	0.03 %
ans		boolean	stack		1 + ↑ + ↑	↑	0	0.00 %
ior	x	boolean	stack		x + x	x	325	0.17 %
ios		boolean	stack		1 + ↑ + ↑	↑	0	0.00 %
xor	x	boolean	stack		x + x	x	0	0.00 %
xos		boolean	stack		1 + ↑ + ↑	↑	0	0.00 %
com	x	boolean	stack		x	x	48	0.03 %
cos		boolean	stack		1 + ↑	↑	0	0.00 %
inn	x	set	stack		1 + x	1	363	0.19 %
ins		set	stack		2 + ↑	1	0	0.00 %
set	x	set	stack		1	x	119	0.06 %
ses		set	stack		2	↑	0	0.00 %
cmu	x	compare	stack		x + x	1	35	0.02 %
cms		compare	stack		1 + ↑ + ↑	1	0	0.00 %
cal	n	call	stack	!	0	0	19147	10.26 %
cas		call	stack	!	1	0	4	0.00 %
ret	n	call	stack	!	???	n	3517	1.88 %
res		call	stack	!	1 + ???	↑	0	0.00 %
beg	x	misc	stack		0	x	1613	0.86 %
bes		misc	stack		1	↑	0	0.00 %
dup	x	misc	stack		x	x + x	79	0.04 %
dus		misc	stack		1	↑ + ↑	0	0.00 %
lor	3	misc	stack	m	0	1	0	0.00 %
str	3	misc	stack	m	1	0	0	0.00 %

3. Mnemonic Table, Sorted by Frequency

This section lists the EM-1 mnemonics. Here they are sorted by static instruction frequency.

Opcode,	Operand	Instr. Group	Exec. Unit	Code	Words Popped	Words Pushed	Static Freq.	Percent
loc	m	load	instr		0	1	27604	14.79 %
cal	n	call	stack	!	0	0	19147	10.26 %
mrk	n	call	mem		0	3	19147	10.26 %
lol	x	load	mem		0	1	17140	9.18 %
lae	x	load	mem		0	a	16242	8.70 %
loe	x	load	mem		0	1	9490	5.08 %
stl	x	store	mem		1	0	6587	3.53 %
ste	x	store	mem		1	0	3588	1.92 %
ret	n	call	stack	!	???	n	3517	1.88 %
lof	x	load	mem		1	1	3472	1.86 %
brf	n	branch	instr	x	0	0	3067	1.64 %
brb	n	branch	instr	x	0	0	2688	1.44 %
lar	x	array	mem		1 + a	#	2111	1.13 %
zeq	n	branch	instr		1	0	2094	1.12 %
cmi		compare	int		2	1	1911	1.02 %
lex	n	load	mem		0	a	1885	1.01 %
lal	x	load	mem		0	a	1805	0.97 %
stf	x	store	mem		2	0	1788	0.96 %
lin	n	misc	instr		0	0	1763	0.94 %
aar	x	array	mem		1 + a	a	1753	0.94 %
bne	n	branch	instr		2	0	1681	0.90 %
beg	x	misc	stack		0	x	1613	0.86 %
ldl	x	load	mem		0	2	1576	0.84 %
loi	y	load	mem		a	y	1561	0.84 %
add		integer	int		2	1	1469	0.79 %
inl	x	inc/dec	mem		0	0	1403	0.75 %
ine	x	inc/dec	mem		0	0	1285	0.69 %
sar	x	array	mem	*	1 + a + #	0	1245	0.67 %
cif		convert	real		1	2	1227	0.66 %
lde	x	load	mem		0	2	1196	0.64 %
teq		compare	int		1	1	1184	0.63 %
zne	n	branch	instr		1	0	1139	0.61 %
lnc	n	load	instr	m	0	1	1136	0.61 %
beq	n	branch	instr		2	0	1060	0.57 %
and	2	boolean	int	m	2	1	1035	0.55 %
blt	n	branch	instr		2	0	1016	0.54 %
rck	x	misc	mem		1	1	987	0.53 %
adi	b	integer	int		1	1	960	0.51 %
sub		integer	int		2	1	878	0.47 %
dup	2	misc	int	m	1	2	845	0.45 %
inc		integer	int		1	1	797	0.43 %
zrl	x	inc/dec	mem		0	0	784	0.42 %
zre	x	inc/dec	mem		0	0	760	0.41 %

Appendix B: EM-1 Instruction and Frequency Tables

Section 3: Mnemonic Table, Sorted by Frequency

Opcode, Operand		Instr. Group	Exec. Unit	Code	Words Popped	Words Pushed	Static Freq.	Percent
sti	y	store	mem	*	a + y	0	746	0.40 %
stp	x	store	mem		1	0	731	0.39 %
ior	2	boolean	int	m	2	1	726	0.39 %
lop	x	load	mem		0	1	676	0.36 %
bgt	n	branch	instr		2	0	598	0.32 %
tne		compare	int		1	1	591	0.32 %
mul		integer	int		2	1	565	0.30 %
blm	x	misc	mem		a + a	0	482	0.26 %
fmu		real	real		4	2	482	0.26 %
dec		integer	int		1	1	478	0.26 %
sde	x	store	mem		2	0	390	0.21 %
div		integer	int		2	1	383	0.21 %
sdl	x	store	mem		2	0	377	0.20 %
inn	x	set	stack		1 + x	1	363	0.19 %
fad		real	real		4	2	355	0.19 %
mod		integer	int		2	1	343	0.18 %
cmf		compare	real		4	1	338	0.18 %
ior	x	boolean	stack		x + x	x	325	0.17 %
ble	n	branch	instr		2	0	305	0.16 %
cfi		convert	real		2	1	303	0.16 %
fsb		real	real		4	2	301	0.16 %
cse	x	misc	instr		1	0	297	0.16 %
bge	n	branch	instr		2	0	293	0.16 %
tlt		compare	int		1	1	292	0.16 %
tgt		compare	int		1	1	290	0.16 %
fdv		real	real		4	2	289	0.15 %
zle	n	branch	instr		1	0	217	0.12 %
tle		compare	int		1	1	172	0.09 %
zge	n	branch	instr		1	0	155	0.08 %
shl		integer	int		2	1	139	0.07 %
tge		compare	int		1	1	134	0.07 %
del	x	inc/dec	mem		0	0	126	0.07 %
set	x	set	stack		1	x	119	0.06 %
neg		integer	int		1	1	108	0.06 %
dup	x	misc	stack		x	x + x	79	0.04 %
ldf	x	load	mem		1	2	73	0.04 %
dee	x	inc/dec	mem		0	0	71	0.04 %
sdf	x	store	mem		a + 1	0	60	0.03 %
and	x	boolean	stack		x + x	x	55	0.03 %
com	x	boolean	stack		x	x	48	0.03 %
zlt	n	branch	instr		1	0	47	0.03 %
cmu	x	compare	stack		x + x	1	35	0.02 %
zgt	n	branch	instr		1	0	31	0.02 %
mon		monitor	instr		0	0	5	0.00 %
cas		call	stack	!	1	0	4	0.00 %
mrs		call	mem		1	3	4	0.00 %
stu		monitor	instr		8	0	0	0.00 %
hlt		misc	instr		0	0	0	0.00 %

Section 3: Mnemonic Table, Sorted by Frequency

Opcode, Operand	Instr. Group	Exec. Unit	Code	Words Popped	Words Pushed	Static Freq.	Percent
str 4	misc	mem	m	1	0	0	0.00 %
str 3	misc	stack	m	1	0	0	0.00 %
str 2	misc	mem	m	1	0	0	0.00 %
str 1	misc	instr	m	1	0	0	0.00 %
str n	misc	instr		1	0	0	0.00 %
lor 4	misc	mem	m	0	1	0	0.00 %
lor 3	misc	stack	m	0	1	0	0.00 %
lor 2	misc	mem	m	0	1	0	0.00 %
lor 1	misc	mem	m	0	1	0	0.00 %
lor n	misc	instr		0	1	0	0.00 %
dus	misc	stack		1	↑+↑	0	0.00 %
bls	misc	mem		1+a+a	0	0	0.00 %
nop	misc	instr	x	0	0	0	0.00 %
bes	misc	stack		1	↑	0	0.00 %
res	call	stack	!	1+???	↑	0	0.00 %
mxs	call	mem	m	1	3	0	0.00 %
mrx n	call	mem	m	0	3	0	0.00 %
cms	compare	stack		1+↑+↑	1	0	0.00 %
cmd	compare	int		4	1	0	0.00 %
aas	array	mem		2+a	a	0	0.00 %
sas	array	mem	*	2+a+#	0	0	0.00 %
las	array	mem		2+a	#	0	0.00 %
ses	set	stack		2	↑	0	0.00 %
ins	set	stack		2+↑	1	0	0.00 %
not	boolean	int		1	1	0	0.00 %
cos	boolean	stack		1+↑	↑	0	0.00 %
xos	boolean	stack		1+↑+↑	↑	0	0.00 %
xor x	boolean	stack		x+x	x	0	0.00 %
ios	boolean	stack		1+↑+↑	↑	0	0.00 %
ans	boolean	stack		1+↑+↑	↑	0	0.00 %
cfd	convert	real		2	2	0	0.00 %
cdf	convert	real		2	2	0	0.00 %
cdi	convert	int		2	1	0	0.00 %
cid	convert	int		1	2	0	0.00 %
dmd	double	int		4	2	0	0.00 %
ddv	double	int		4	2	0	0.00 %
dmu	double	int		4	2	0	0.00 %
dsb	double	int		4	2	0	0.00 %
dad	double	int		4	2	0	0.00 %
ads	integer	int		2	1	0	0.00 %
exg	integer	int		2	2	0	0.00 %
ror	integer	int		2	1	0	0.00 %
rol	integer	int		2	1	0	0.00 %
shr	integer	int		2	1	0	0.00 %
sts	store	mem	*	1+a+↑	0	0	0.00 %
sib	store	mem	m	a+1	0	0	0.00 %
los	load	mem		1+a	↑	0	0.00 %
lib	load	mem	m	a	1	0	0.00 %

4. Mnemonic Table, Sorted by Mnemonic

This section lists the EM-1 mnemonics. Here the mnemonics are listed in alphabetical order.

Opcode, Operand		Instr. Group	Exec. Unit	Code	Words Popped	Words Pushed	Static Freq.	Percent
aar	x	array	mem		1+a	a	1753	0.94 %
aas		array	mem		2+a	a	0	0.00 %
add		integer	int		2	1	1469	0.79 %
adi	b	integer	int		1	1	960	0.51 %
ads		integer	int		2	1	0	0.00 %
and	x	boolean	stack		x+x	x	55	0.03 %
and	2	boolean	int	m	2	1	1035	0.55 %
ans		boolean	stack		1+↑+↑	↑	0	0.00 %
beg	x	misc	stack		0	x	1613	0.86 %
beq	n	branch	instr		2	0	1060	0.57 %
bes		misc	stack		1	↑	0	0.00 %
bge	n	branch	instr		2	0	293	0.16 %
bgt	n	branch	instr		2	0	598	0.32 %
ble	n	branch	instr		2	0	305	0.16 %
blm	x	misc	mem		a+a	0	482	0.26 %
bls		misc	mem		1+a+a	0	0	0.00 %
blt	n	branch	instr		2	0	1016	0.54 %
bne	n	branch	instr		2	0	1681	0.90 %
brb	n	branch	instr	x	0	0	2688	1.44 %
brf	n	branch	instr	x	0	0	3067	1.64 %
cal	n	call	stack	!	0	0	19147	10.26 %
cas		call	stack	!	1	0	4	0.00 %
cdf		convert	real		2	2	0	0.00 %
cdi		convert	int		2	1	0	0.00 %
cfd		convert	real		2	2	0	0.00 %
cfi		convert	real		2	1	303	0.16 %
cid		convert	int		1	2	0	0.00 %
cif		convert	real		1	2	1227	0.66 %
cmd		compare	int		4	1	0	0.00 %
cmf		compare	real		4	1	338	0.18 %
cmi		compare	int		2	1	1911	1.02 %
cms		compare	stack		1+↑+↑	1	0	0.00 %
cmu	x	compare	stack		x+x	1	35	0.02 %
com	x	boolean	stack		x	x	48	0.03 %
cos		boolean	stack		1+↑	↑	0	0.00 %
cse	x	misc	instr		1	0	297	0.16 %
dad		double	int		4	2	0	0.00 %
ddv		double	int		4	2	0	0.00 %
dec		integer	int		1	1	478	0.26 %
dee	x	inc/dec	mem		0	0	71	0.04 %
del	x	inc/dec	mem		0	0	126	0.07 %
div		integer	int		2	1	383	0.21 %
dmd		double	int		4	2	0	0.00 %

Appendix B: EM-1 Instruction and Frequency Tables

Section 4: Mnemonic Table, Sorted by Mnemonic

Opcode, Operand		Instr. Group	Exec. Unit	Code	Words Popped	Words Pushed	Static Freq.	Percent
dmu		double	int		4	2	0	0.00 %
dsb		double	int		4	2	0	0.00 %
dup	x	misc	stack		x	x + x	79	0.04 %
dup	2	misc	int	m	1	2	845	0.45 %
dus		misc	stack		1	↑ + ↑	0	0.00 %
exg		integer	int		2	2	0	0.00 %
fad		real	real		4	2	355	0.19 %
fdv		real	real		4	2	289	0.15 %
fmu		real	real		4	2	482	0.26 %
fsb		real	real		4	2	301	0.16 %
hlt		misc	instr		0	0	0	0.00 %
inc		integer	int		1	1	797	0.43 %
ine	x	inc/dec	mem		0	0	1285	0.69 %
inl	x	inc/dec	mem		0	0	1403	0.75 %
inn	x	set	stack		1 + x	1	363	0.19 %
ins		set	stack		2 + ↑	1	0	0.00 %
ior	x	boolean	stack		x + x	x	325	0.17 %
ior	2	boolean	int	m	2	1	726	0.39 %
ios		boolean	stack		1 + ↑ + ↑	↑	0	0.00 %
lae	x	load	mem		0	a	16242	8.70 %
lal	x	load	mem		0	a	1805	0.97 %
lar	x	array	mem		1 + a	#	2111	1.13 %
las		array	mem		2 + a	#	0	0.00 %
lde	x	load	mem		0	2	1196	0.64 %
ldf	x	load	mem		1	2	73	0.04 %
ldl	x	load	mem		0	2	1576	0.84 %
lex	n	load	mem		0	a	1885	1.01 %
lib		load	mem	m	a	1	0	0.00 %
lin	n	misc	instr		0	0	1763	0.94 %
lnc	n	load	instr	m	0	1	1136	0.61 %
loc	m	load	instr		0	1	27604	14.79 %
loe	x	load	mem		0	1	9490	5.08 %
lof	x	load	mem		1	1	3472	1.86 %
loi	y	load	mem		a	y	1561	0.84 %
lol	x	load	mem		0	1	17140	9.18 %
lop	x	load	mem		0	1	676	0.36 %
lor	n	misc	instr		0	1	0	0.00 %
lor	1	misc	mem	m	0	1	0	0.00 %
lor	2	misc	mem	m	0	1	0	0.00 %
lor	3	misc	stack	m	0	1	0	0.00 %
lor	4	misc	mem	m	0	1	0	0.00 %
los		load	mem		1 + a	↑	0	0.00 %
mod		integer	int		2	1	343	0.18 %
mon		monitor	instr		0	0	5	0.00 %
mrk	n	call	mem		0	3	19147	10.26 %
mrs		call	mem		1	3	4	0.00 %
mrx	n	call	mem	m	0	3	0	0.00 %
mul		integer	int		2	1	565	0.30 %

Appendix B: EM-1 Instruction and Frequency Tables

Section 4: Mnemonic Table, Sorted by Mnemonic

Opcode,		Instr.	Exec.		Words	Words	Static	
Operand		Group	Unit	Code	Popped	Pushed	Freq.	Percent
mxs		call	mem	m	1	3	0	0.00 %
neg		integer	int		1	1	108	0.06 %
nop		misc	instr	x	0	0	0	0.00 %
not		boolean	int		1	1	0	0.00 %
rck	x	misc	mem		1	1	987	0.53 %
res		call	stack	!	1 + ???	↑	0	0.00 %
ret	n	call	stack	!	???	n	3517	1.88 %
rol		integer	int		2	1	0	0.00 %
ror		integer	int		2	1	0	0.00 %
sar	x	array	mem	*	1 + a + #	0	1245	0.67 %
sas		array	mem	*	2 + a + #	0	0	0.00 %
sde	x	store	mem		2	0	390	0.21 %
sdf	x	store	mem		a + 1	0	60	0.03 %
sdl	x	store	mem		2	0	377	0.20 %
ses		set	stack		2	↑	0	0.00 %
set	x	set	stack		1	x	119	0.06 %
shl		integer	int		2	1	139	0.07 %
shr		integer	int		2	1	0	0.00 %
sib		store	mem	m	a + 1	0	0	0.00 %
ste	x	store	mem		1	0	3588	1.92 %
stf	x	store	mem		2	0	1788	0.96 %
sti	y	store	mem	*	a + y	0	746	0.40 %
stl	x	store	mem		1	0	6587	3.53 %
stp	x	store	mem		1	0	731	0.39 %
str	n	misc	instr		1	0	0	0.00 %
str	1	misc	instr	m	1	0	0	0.00 %
str	2	misc	mem	m	1	0	0	0.00 %
str	3	misc	stack	m	1	0	0	0.00 %
str	4	misc	mem	m	1	0	0	0.00 %
sts		store	mem	*	1 + a + ↑	0	0	0.00 %
stu		monitor	instr		8	0	0	0.00 %
sub		integer	int		2	1	878	0.47 %
teq		compare	int		1	1	1184	0.63 %
tge		compare	int		1	1	134	0.07 %
tgt		compare	int		1	1	290	0.16 %
tle		compare	int		1	1	172	0.09 %
tlt		compare	int		1	1	292	0.16 %
tne		compare	int		1	1	591	0.32 %
xor	x	boolean	stack		x + x	x	0	0.00 %
xos		boolean	stack		1 + ↑ + ↑	↑	0	0.00 %
zeq	n	branch	instr		1	0	2094	1.12 %
zge	n	branch	instr		1	0	155	0.08 %
zgt	n	branch	instr		1	0	31	0.02 %
zle	n	branch	instr		1	0	217	0.12 %
zlt	n	branch	instr		1	0	47	0.03 %
zne	n	branch	instr		1	0	1139	0.61 %
zre	x	inc/dec	mem		0	0	760	0.41 %
zrl	x	inc/dec	mem		0	0	784	0.42 %

5. Opcode Frequency Table, Sorted by Opcode

This section contains a list of the 492 most frequent mnemonic and operand combinations. The list is sorted alphabetically by opcode. The operands are divided into high order byte and low order byte. The byte values are given in decimal.

		Operand	
Freq	Opcode	High	Low
47	aar	0	143
100	aar	1	58
43	aar	1	130
43	aar	1	210
43	aar	1	211
48	aar	17	22
44	aar	19	35
44	aar	19	38
44	aar	28	62
75	aar	45	206
63	aar	45	209
75	aar	60	187
1469	add	-	-
71	adi	0	1
191	adi	0	2
61	adi	0	6
41	adi	0	8
74	adi	0	16
71	adi	0	18
1035	and	0	1
52	and	0	4
464	beg	0	1
372	beg	0	2
198	beg	0	3
147	beg	0	4
58	beg	0	5
86	beg	0	6
45	beg	0	7
129	beq	0	2
213	beq	0	3
50	beq	0	4
65	beq	0	5
60	beq	0	6
55	bgt	0	12
158	blm	0	8
87	blm	0	10
50	blt	0	11
55	blt	0	12
87	bne	0	2
83	bne	0	3
196	bne	0	4
163	bne	0	5
224	bne	0	6
74	bne	0	7
55	bne	0	8
45	bne	0	9
66	bne	0	10
41	bne	0	12
55	brb	0	12
53	brb	0	13
53	brb	0	14
95	brb	0	15
85	brb	0	16
72	brb	0	17
78	brb	0	18
65	brb	0	19
68	brb	0	20
52	brb	0	21
48	brb	0	22
66	brb	0	23
67	brb	0	24
48	brb	0	25
54	brb	0	26
45	brb	0	27
43	brb	0	28
42	brb	0	29
46	brb	0	30
50	brb	0	31
47	brb	0	35
43	brb	0	36
181	brf	0	2
130	brf	0	3
130	brf	0	4
102	brf	0	5
88	brf	0	6
59	brf	0	7
76	brf	0	8
68	brf	0	9
67	brf	0	10
62	brf	0	11
58	brf	0	12
46	brf	0	14
40	brf	0	16
46	brf	0	18
47	brf	0	22
1202	cal	0	1
1040	cal	0	2
969	cal	0	3
1386	cal	0	4
962	cal	0	5
940	cal	0	6
761	cal	0	7
1179	cal	0	8
618	cal	0	9
696	cal	0	10
956	cal	0	11
651	cal	0	12
390	cal	0	13
423	cal	0	14
446	cal	0	15
508	cal	0	16
315	cal	0	17
221	cal	0	18
305	cal	0	19
246	cal	0	20
211	cal	0	21
221	cal	0	22
209	cal	0	23
222	cal	0	24
242	cal	0	25
171	cal	0	26
153	cal	0	27
121	cal	0	28
133	cal	0	29
122	cal	0	30
113	cal	0	31
105	cal	0	32
98	cal	0	33
124	cal	0	34
79	cal	0	35
98	cal	0	36
72	cal	0	37
70	cal	0	38
66	cal	0	39
54	cal	0	40
160	cal	0	41
72	cal	0	42
47	cal	0	43
40	cal	0	44
55	cal	0	46
48	cal	0	47
288	cal	0	50
61	cal	0	51
51	cal	0	52
45	cal	0	55
41	cal	0	56
51	cal	0	59
58	cal	0	61
62	cal	0	67
72	cal	0	78
59	cal	0	79
77	cal	0	169
303	cfi	-	-
1227	cif	-	-
338	cmf	-	-
1911	cmi	-	-
45	com	0	4
478	dec	-	-

Appendix B: EM-1 Instruction and Frequency Tables

Section 5: Opcode Frequency Table, Sorted by Opcode

		Operand	
Freq	Opcode	High	Low
383	div	-	-
845	dup	0	1
68	dup	0	2
355	fad	-	-
289	fdv	-	-
482	fmu	-	-
301	fsb	-	-
797	inc	-	-
529	ine	0	0
43	ine	0	5
90	ine	0	6
50	ine	0	7
43	ine	0	14
313	inl	0	0
326	inl	0	1
200	inl	0	2
151	inl	0	3
106	inl	0	4
67	inl	0	5
41	inl	0	6
272	inn	0	4
726	ior	0	1
322	ior	0	4
846	lae	0	1
5681	lae	0	3
870	lae	0	5
149	lae	0	6
164	lae	0	7
165	lae	0	8
210	lae	0	9
193	lae	0	10
253	lae	0	11
178	lae	0	12
111	lae	0	13
63	lae	0	14
55	lae	0	15
59	lae	0	17
79	lae	0	19
66	lae	0	20
44	lae	0	22
41	lae	0	28
145	lae	0	45
155	lae	0	59
42	lae	0	69
48	lae	0	85
41	lae	0	87
343	lae	0	105
57	lae	0	148
48	lae	0	171
49	lae	0	187
57	lae	1	2
92	lae	1	9
134	lae	1	149
44	lae	2	115
75	lae	25	127
329	lal	0	0
136	lal	0	1
178	lal	0	2
118	lal	0	3
102	lal	0	4
133	lal	0	5
114	lal	0	6
83	lal	0	7
101	lal	0	8
50	lal	0	9
60	lal	0	10
56	lar	0	107
62	lar	0	147
42	lar	12	122
113	lar	26	193
44	lar	26	208
83	lde	0	13
42	lde	0	17
491	ldl	0	0
236	ldl	0	1
188	ldl	0	2
84	ldl	0	3
108	ldl	0	4
145	ldl	0	6
98	ldl	0	8
49	ldl	0	12
1696	lex	0	1
143	lex	0	2
3012	loc	0	0
4470	loc	0	1
1764	loc	0	2
1276	loc	0	3
1066	loc	0	4
822	loc	0	5
1274	loc	0	6
582	loc	0	7
942	loc	0	8
573	loc	0	9
1557	loc	0	10
285	loc	0	11
249	loc	0	12
405	loc	0	13
171	loc	0	14
242	loc	0	15
227	loc	0	16
163	loc	0	17
147	loc	0	18
121	loc	0	19
212	loc	0	20
148	loc	0	21
120	loc	0	22
96	loc	0	23
114	loc	0	24
181	loc	0	25
81	loc	0	26
121	loc	0	27
84	loc	0	28
71	loc	0	29
113	loc	0	30
59	loc	0	31
626	loc	0	32
103	loc	0	33
59	loc	0	34
45	loc	0	35
68	loc	0	36
68	loc	0	39
128	loc	0	40
84	loc	0	41
149	loc	0	42
56	loc	0	43
109	loc	0	44
122	loc	0	45
183	loc	0	46
65	loc	0	47
226	loc	0	48
91	loc	0	50
50	loc	0	51
51	loc	0	52
62	loc	0	54
46	loc	0	55
45	loc	0	56
93	loc	0	57
89	loc	0	58
68	loc	0	60
69	loc	0	61
42	loc	0	62
73	loc	0	63
98	loc	0	64
64	loc	0	80
45	loc	0	92
120	loc	0	97
55	loc	0	99
215	loc	0	100
79	loc	0	101
41	loc	0	106
44	loc	0	108
55	loc	0	110
53	loc	0	116
64	loc	0	122
54	loc	0	125
42	loc	0	128
117	loc	1	0
58	loc	3	232
48	loc	4	0
1136	loc	255	255
46	loe	0	2
363	loe	0	5
292	loe	0	6
711	loe	0	7
378	loe	0	8
206	loe	0	9
115	loe	0	10
156	loe	0	11
117	loe	0	12
318	loe	0	13
236	loe	0	14
89	loe	0	15
123	loe	0	16

Appendix B: EM-1 Instruction and Frequency Tables

Section 5: Opcode Frequency Table, Sorted by Opcode

		Operand	
Freq	Opcode	High	Low
105	loe	0	17
64	loe	0	18
278	loe	0	19
66	loe	0	20
65	loe	0	21
67	loe	0	22
81	loe	0	23
56	loe	0	24
42	loe	0	26
53	loe	0	29
72	loe	0	30
71	loe	0	49
58	loe	0	50
78	loe	0	62
62	loe	0	72
68	loe	0	97
58	loe	0	102
80	loe	0	106
96	loe	0	107
65	loe	0	109
52	loe	0	110
122	loe	0	113
89	loe	0	114
56	loe	0	117
77	loe	0	119
157	loe	0	126
57	loe	0	138
69	loe	0	246
48	loe	1	103
58	loe	12	48
52	loe	15	182
42	loe	19	21
68	loe	21	27
810	lof	0	1
630	lof	0	2
330	lof	0	3
241	lof	0	4
168	lof	0	5
163	lof	0	6
134	lof	0	7
92	lof	0	8
104	lof	0	9
68	lof	0	10
70	lof	0	11
45	lof	0	12
61	lof	0	13
47	lof	0	14
47	lof	0	15
43	lof	0	16
43	lof	9	200
798	loi	0	1
641	loi	0	4
4391	lol	0	0
3034	lol	0	1
2208	lol	0	2
1707	lol	0	3
1202	lol	0	4
969	lol	0	5
729	lol	0	6
559	lol	0	7
439	lol	0	8
310	lol	0	9
280	lol	0	10
222	lol	0	11
142	lol	0	12
128	lol	0	13
83	lol	0	14
108	lol	0	15
53	lol	0	16
52	lol	0	17
292	lop	0	0
112	lop	0	1
81	lop	0	2
343	mod	-	-
11111	mrk	0	0
6144	mrk	0	1
1183	mrk	0	2
384	mrk	0	3
152	mrk	0	4
143	mrk	0	5
565	mul	-	-
108	neg	-	-
45	rck	5	138
160	rck	5	140
109	rck	5	142
46	rck	5	163
51	rck	60	202
3145	ret	0	0
316	ret	0	1
56	ret	0	2
41	sar	28	59
67	sdl	0	4
89	sdl	0	6
118	set	0	4
139	shl	-	-
157	ste	0	5
167	ste	0	6
235	ste	0	7
150	ste	0	8
110	ste	0	9
48	ste	0	10
46	ste	0	11
63	ste	0	13
70	ste	0	14
69	ste	0	19
44	ste	0	97
52	ste	0	114
42	ste	0	119
64	ste	0	126
344	stf	0	1
333	stf	0	2
144	stf	0	3
97	stf	0	4
70	stf	0	5
116	stf	0	6
111	stf	0	7
86	stf	0	8
64	stf	0	9
569	sti	0	1
145	sti	0	4
872	stl	0	0
1073	stl	0	1
1185	stl	0	2
728	stl	0	3
543	stl	0	4
429	stl	0	5
339	stl	0	6
258	stl	0	7
204	stl	0	8
152	stl	0	9
89	stl	0	10
102	stl	0	11
66	stl	0	12
48	stl	0	13
56	stl	0	14
43	stl	0	15
164	stp	0	0
68	stp	0	1
293	stp	0	2
51	stp	0	3
43	stp	0	4
878	sub	-	-
1184	teq	-	-
134	tge	-	-
290	tgt	-	-
172	tle	-	-
292	tlt	-	-
591	tne	-	-
65	zeq	0	1
135	zeq	0	2
221	zeq	0	3
169	zeq	0	4
125	zeq	0	5
100	zeq	0	6
85	zeq	0	7
49	zeq	0	8
43	zeq	0	9
59	zeq	0	10
61	zeq	0	11
42	zeq	0	19
271	zne	0	2
68	zne	0	3
67	zne	0	4
50	zne	0	5
47	zne	0	6
52	zne	0	7
107	zrl	0	0
115	zrl	0	1
86	zrl	0	2
93	zrl	0	3
54	zrl	0	4
82	zrl	0	5
67	zrl	0	6