An Extensible, scalable microprocessor architecture

Matthew Dinmore

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation


This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
An Extensible, Scalable Microprocessor Architecture

by

Matthew Dinmore

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE in Computer Engineering

Approved by:

Thesis Advisor Dr. Tony Chang

Department Head Dr. Roy Czernikowski

Committee Member Prof. George Brown

DEPARTMENT OF COMPUTER ENGINEERING COLLEGE OF ENGINEERING ROCHESTER INSTITUTE OF TECHNOLOGY ROCHESTER, NEW YORK NOVEMBER, 1993
Title of Thesis: An Extensible, Scalable Microprocessor Architecture

I, Matthew D. Dinmore, hereby grant permission to the Wallace Memorial Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part.

Date: 30 Nov 93
Abstract

An extensible, scalable stack-based microprocessor architecture is developed and discussed. Several unique features of the architecture, including its non-memory oriented interface, and its use of a stack for holding and executing code, are detailed. A programmed model is used to verify the architecture, and a hardware implementation of a small-scale version of the architecture is constructed and tested. Notes for future implementations are provides. Possible applications based on the latest technological trends are discussed, and topics for further research into the architecture are listed.
## Contents

Abstract .............................................................................................................. i

List of Diagrams ................................................................................................. v

List of Tables ........................................................................................................ vi

List of Acronyms & Abbreviations ................................................................. vii

1 • Introduction .................................................................................................... 1
  1.1.0 Background & Goals ............................................................................. 1
  1.2.0 Definitions ............................................................................................ 2
  1.3.0 Overview ............................................................................................... 3

2 • Design Issues ................................................................................................ 4
  2.0.0 Basic Services ....................................................................................... 4
  2.1.0 Input/Output Performance .................................................................. 6
  2.2.0 System Interfaces ................................................................................. 8
  2.3.0 Program Execution Model .................................................................... 10
  2.4.0 Additional Architectural Issues ......................................................... 14
    2.4.1 Stack Sizes ....................................................................................... 14
    2.4.2 Interrupts ............................................................................................ 16
    2.4.3 Levels of Implementation ................................................................. 19
  2.5.0 Instruction Set Design ......................................................................... 22
    2.5.1 Arithmetic & Logic Instructions ....................................................... 23
    2.5.2 Stack Manipulation Operations ....................................................... 28
    2.5.3 Input/Output Instructions ................................................................ 32
    2.5.4 Control Operations ............................................................................ 33
2.5.5 Program Execution Instructions ................................... 35
2.5.6 Extended Instructions ............................................... 37
2.5.7 Instruction Encoding ................................................ 38

2.6.0 Instruction Mandates for Each Implementation Level ..... 45

3 • Implementation & Testing ............................................. 47

3.0.0 Overview ......................................................................................... 47
3.1.0 VHDL Model ............................................................................... 48

3.1.1 Entity Definition ......................................................................... 49
3.1.2 Model Internals ........................................................................... 50
3.1.3 VHDL Test Bench ......................................................................... 51

3.2.0 FPGA Implementation ............................................................. 53

3.2.1 FPGA Technology ....................................................................... 53
3.2.2 The Processor Design ................................................................. 54
3.2.3 Overall Design ........................................................................... 56

3.2.4 Arithmetic/Logic Unit (ALU) ..................................................... 57
3.2.5 Input/Output Unit (IOU) ............................................................. 60
3.2.6 Data Stack (DS) .......................................................................... 61
3.2.7 Control Register Block (CRB) .................................................. 62
3.2.8 Control Unit (CU) ....................................................................... 64
3.2.9 Additional Design Notes ............................................................ 69

3.2.10 Design Testing ........................................................................... 69
3.2.11 FPGA Test Bench ....................................................................... 72

3.3.0 General Implementation Notes ................................................. 74
List of Diagrams

2.1 I/O Protocol ................................................................. 9
2.2 Architectural Levels ...................................................... 20
2.3 Architectural Layers of Definition .............................. 21
3.1 VHDL Model Design ..................................................... 49
3.2 System Layout ............................................................. 57
3.3 Arithmetic/Logic Bit Slice ............................................. 58
3.4 Input Handshake Timing ............................................... 60
3.5 Output Handshake Timing .............................................. 61
3.6 Cycle Timing ................................................................. 65
3.7 Processor Pin Placement .............................................. 72
3.8 Processor/Switch Board Test Bench ............................. 73
3.9 Processor/MC68008 Test Bench ................................. 74
3.10 Dynamic Power Consumption ................................. 78
List of Tables

2.1 Instruction Set ......................................................... 43-44
3.1 ALU Internal Control Signals ....................................... 59
3.2 Control Signal Generation ............................................ 66
3.3 Pin Requirements for Various Processor Scales ............. 76
List of Acronyms & Abbreviations

ALU ........................................... Arithmetic and Logic Unit
ASCII ........................................ American Standard Code for Information Interchange
ASIC .......................................... Application-Specific Integrated Circuit
CISC .............................................. Complex Instruction Set Computer
CRB ............................................... Control Register Block
CU ................................................ Control Unit
DS ................................................ Data Stack
DSP ................................................ Data Stack Pointer
FPGA ............................................. Field Programmable Gate Array
I/O ............................................... Input/Output
IB ................................................ Instruction Bit
IOCR ............................................. Input/Output Control Register
IOU ............................................... Input/Output Unit
IS ................................................ Instruction Stack
ISP ................................................ Instruction Stack Pointer
IXA ............................................... Implementation Extensible Architecture
LIFO ............................................. Last-In, First Out (a stack)
LSB ............................................... Least-Significant Bit
MIPS ............................................ Millions of Instructions Per Second
MSB ............................................... Most-Significant Bit
PC ............................................... Program Controller
PDA ............................................... Personal Digital Assistant
RISC ............................................ Reduced Instruction Set Computer
RPN .............................................. Reverse Polish Notation

SIMD ........................................... Single Instruction, Multiple Data

SMI ............................................. Standard Memory Interface

SP ............................................... Stack Pointer

SR ............................................... Status Register

SW ............................................... Status Word

TOS ............................................. Top Of Stack

VHDL .......................................... VHSIC Hardware Description Language

VHSIC .......................................... Very High Speed Integrated Circuit

VLSI ............................................ Very Large Scale Integration
1 • Introduction

1.1.0 Background & Goals

The proliferation of low-cost microprocessors has made the task of selecting a processor for a particular application more complicated. System implementers may choose a general-purpose processor which provides the functionality necessary for the application, or they can design a custom processor. By selecting an off-the-shelf system, costs are kept low, time is saved, and a certain level of compatibility is established. However, the processor is not necessarily optimized for the application, and there may be features which go completely unused. Developing a custom processor would alleviate these problems, but would be expensive in terms of time and money, and would lack compatibility with existing systems and tools. A better solution would be to base a new system on a simple, standard processor which provides a core set of computing services and excellent scalability and extensibility. This thesis develops and demonstrates an architecture which incorporates these basic goals.

Additionally, the architecture has been designed such that it is fabrication technology independent. In modern microprocessor design, there are two approaches to improving performance; the fabrication technology used to produce the processor can be improved, or the internal, logical architecture of the processor can be changed. Certainly, the two approaches can be used in conjunction to
further performance, but doing so ties logical features of the architecture to the technology, a situation which could impede implementation of the architecture in other technologies. The architecture developed here is implementation-technology independent to allow its fabrication in any technology, a feature which is demonstrated.

1.2.0 Definitions

In this context, scalability refers to the fact that design variables such as the word size, amount of internal storage or buffering, and portion of the defined instruction set which is implemented in the processor are not rigidly set by the architecture. Instead, these variables would be determined by the processor or system implementer to meet the needs of the application.

Extensibility refers to the architecture's ability to support implementation-specific extensions to the instruction set or supporting hardware. This allows the system designer to take the basic architecture and add capabilities supporting the specific application.
1.3.0 Overview

In order to provide a basis for the development of the architecture, several issues of processor design are discussed relative to current solutions. From this discussion, various aspects of the architecture are introduced. A detailed discussion of the architecture follows. To put the architecture in perspective, it is compared to some similar systems.

A VHDL model of one possible implementation of the architecture has been created and used to test various aspects of the architecture. A different implementation has been performed in hardware to demonstrate the architecture's implementation flexibility. Both of these implementations are detailed and discussed.

Finally, the unique nature of the architecture requires that topics such as programming (and the software tools to support development), external interfacing and implementation will be discussed. Applications for the architecture will be briefly outlined.
2 • Design Issues

2.0.0 Basic Services

The purpose of developing a new architecture is to provide solutions to computing problems. As discussed above, an application-specific solution has the drawback of being limited to just one purpose, a restriction which may not prove economical against the high cost of producing a new processor. Since the goal of this architecture is to provide basic computing services while not imposing the restrictions on design found in either general-purpose or application-specific processors, it is necessary to define what these basic services are.

All computing units must be able to accept data and instructions, apply the instructions to the data, and produce results. To be able to accept input and produce output, the processor must have an interface between its internal and external environment. The specification of this interface is critical to the systems implementer, because it determines what types of external devices can be connected to the processor, and what protocol they will use to communicate. To make the architecture as extensible as possible, a simple protocol and interface scheme is required.

The selection of instructions for data processing will determine what types of computational problems the processor can efficiently handle. Early in computer...
development, complex instruction set computers (CISC) provided instructions to handle a wide variety of computing tasks. The belief was that by providing powerful and flexible instructions in hardware, programmers would be able to spend more time writing applications, and less time writing subroutines to perform basic tasks. CISC machines provided this capability at the expense of complex decoding and instruction execution mechanisms. As programmers became more reliant on compilers than hand assembly, fewer of the complex instructions were used, since the compiler tends to convert source code in the most generic manner to improve efficiency. As such, if the programmer wrote a routine in a high-level language to perform the same function as an existing processor instruction, the compiler would assemble it from simpler instructions rather than use the single, more complex instruction.

The realization of this fact lead computer architects to devise reduced instruction set computers (RISC), an architectural paradigm in which only basic instructions with few addressing modes are provided. The task of creating more complex instructions is left to the compiler. The instruction sets of these computers provide a good model for the development of a basic instruction set.
2.1.0 Input/Output Performance

The first task is for the processor to be able to input data and instructions. The optimal way to do this is to be able to process a continuous stream of data and instructions while producing a continuous output stream of results. Some systems which attempt to do this include systolic arrays and certain digital signal processors. Both of these rely on the regularity of the data and consistency of the instruction execution cycle to achieve high performance. However, this is generally not possible since instruction execution is often conditional, and data may reside in the processor’s internal or the system’s external memory in an almost random order.

Systolic arrays are able to achieve high performance by repeatedly executing an instruction sequence on data which is continuously shifted through each processor in the array [3, §2.18]. In this mode of operation, the instruction sequence is either hard-wired into the processor, or loaded before data processing begins (assuming a reconfigurable array, of course). Once processing begins, no special instructions are required to read or write data; it simply ‘flows’ in and out of the processor. This is, in a sense, an implicit addressing scheme. This model does not allow for conditional code execution or any change in the code sequence during execution, thereby reducing the number of applications for which the array is useful to a very few. However, it does demonstrate the power of implicit addressing. In
order to make the process more generic, it is necessary to provide a means of loading and internally storing data, and then operating on the data with instruction sequences which are not necessarily static for the duration of the computing task.

This can be accomplished by using a stack. This is a last-in, first-out (LIFO) data structure which requires no addressing or special instructions for loading data (unlike the register-based model commonly found in CISC and RISC processors). Instead, it is only necessary for the input system to be able to differentiate data and instructions, so that they can share the same input/output (I/O) pathways, an important consideration both in terms of logical integration and the physical limitations of implementation. All that is required to do this is a single bit. If we consider this bit to be set when representing an instruction and cleared when representing data, then it is possible to load data and process it from a single stream of words from an external source; the nature of this source is irrelevant to the function of the processor. Any word arriving at the input with a cleared instruction bit (the formal name for the flag bit) is implicitly pushed onto the stack, while a word arriving with a set instruction bit is directed to the execution unit.
2.2.0 System Interfaces

The goal in designing the interface to the processor is to provide the lowest common denominator upon which more powerful interfaces can be constructed. Two elements are required; a data path and a means of providing control information, i.e. handshaking between the components.

The data path may be either serial or parallel. Serial systems provide simplicity in terms of interconnection, but are slower than parallel systems. Additionally, since processors generally handle data in parallel mode internally, less hardware is required to implement the interface.

To provide handshaking between systems, a standard arrangement requiring handshake input and output signals on both devices can be used. The protocol for handling communications is the important consideration, and needs to be verified for proper operation. A Petri diagram with initial markings for the reset (no I/O occurring) state is presented below (Diagram 2.1):
The protocol calls for the sender to raise its output handshake (IHᵢ) and hold it while the data on the bus is valid (the data should actually be valid prior to the assertion of the handshake so that the receiver can use the handshake to latch the data into an input buffer, if necessary). The sender must continue to hold its handshake until the receiver replies with its own handshake (IHₒ). The receiver then holds its acknowledgment handshake until the sender drops its handshake, after which the receiver can drop its own.
This protocol can be used on both uni- and bidirectional busses. With a unidirectional bus, there is no possibility of collision, so the protocol need not address collisions. In the event the bus is bidirectional, some type of collision detection is required. In keeping with the goal of providing only the most basic definition in the architecture, no collision detection or handling protocol will be specified. If an implementation requires a more robust system, one can be implemented externally. To mandate a protocol would increase complexity for all cases, even those for which collisions are not a possibility.

Note that the protocol is not deadlock-free. If either party fails to respond to a handshake, the system will freeze. Again, defining a protocol to handle lock-up increases complexity for all implementations, even those where deadlock is not an issue. Instead, deadlock should be considered a system error condition.

2.3.0 Program Execution Model
As discussed above, data and instructions can be loaded into the processor using the same protocols. Using a stack, data can first be loaded and then processed by instructions from the same incoming stream of words. This format for expression handling is known as Reverse Polish Notation (RPN), and is the most efficient way to evaluate arbitrarily complex expressions. The question of how the instructions are added to the data stream remains to be discussed.
Standard processors begin fetching instructions from memory when they are started. These instructions are located in blocks in memory, and the instructions in each block are sequential. The processor is able to branch or jump within and between blocks by executing specific branching operations, often based on the current state of the processor. Since no memory interface has been defined in this architecture, this process is not directly possible. Instead, this architecture relies on an external program controller (PC) to feed instructions and data to the processor in an executable stream. The nature of this controller is not specified, since it will define what the system does. However, the interface to the processor for the controller is clearly defined and standard, so that processors and controllers can be interchanged.

With such a controller, a standard memory interface could be created and programs in memory could be executed as with any other processor. The controller would handle reading and writing data to memory, and would be able to perform branches and jumps. This is made possible by allowing additional instructions to be defined (the architecture's extensibility), and by providing the processor's internal status to the PC though a set of dedicated signals.

Building a system in this way has no clear benefits over a traditional processor; in fact, if the processor and the PC are physically separated, performance would certainly be much less efficient than a traditional system. The main reason for
this is the time required to execute branches. This is one of the largest problems in any architecture, but even more so here due to the extra time required to communicate the processor’s status before a conditional branch.

Recent processors often address this issue through the use of branch prediction. A prediction about the outcome of the branch is made, often with the help of information encoded in the instruction which indicates the most-often taken path (this is the method used in processors such as the IBM/Motorola PowerPC [7, §1.4.1]). If the prediction succeeds, nothing is lost in execution, while if it fails, the processor’s pipeline often needs to be flushed causing a decrease in performance.

This architecture does not allow for easy pipelining because of the inherent data dependency caused by the use of a stack. However, conditional branches will still lower performance. To improve the situation, it would be useful to move code fetch and execution inside the processor under certain circumstances. This can be accomplished by providing a means of loading a code segment, which will possibly be repeatedly executed, into the processor, and then executing it without interaction from the external controller.

Since there is no addressing scheme built into the processor, it makes sense to implement a second stack which will be used to hold the code as it is loaded. A special instruction is required to begin the load process (otherwise incoming
instructions are immediately executed); execution of this instruction places the processor into a mode which directs incoming instructions to the second stack (the Instruction Stack or IS) until an escape instruction is executed to return the processor to normal mode. Note that incoming instructions are still tagged with the instruction bit and that data can still be loaded concurrently with instructions, since it will be directed to the Data Stack (DS).

Once the IS has been loaded, instruction execution can be switched from the I/O ports to the IS. Instructions are then executed from the IS until an instruction is reached which stops IS-based execution. During execution, the IS behaves like a stack only in that the stack pointer is used to locate the next instruction. Instructions for modifying the stack pointer, and thereby allowing branching and looping, are necessary.

Instructions executed from the IS act upon the DS in the same manner as instructions coming from the I/O system. Data can also still be loaded implicitly during stack-based execution by passing it to the I/O system as during normal execution; an arriving word is treated as the top-of-stack (TOS) item, as it would be if it were pushed onto the DS. This allows continuous data processing without any explicit data-load instructions being executed, a clear advantage over standard processors for certain applications. However, an instruction to perform an explicit data-load operations is also required for programmed reading of data.
2.4.0 Additional Architectural Issues

The above discussion provides a core definition for an extensible, scalable, stack-based architecture for which the name Implementation Extensible Architecture (IXA) has been chosen.

The remainder of this section discusses some other aspects of the architecture which are less critical to the overall definition, yet require some discussion. A complete, stand-alone definition for the architecture is provided in Appendix A.

2.4.1 Stack Sizes

The size of either of the internal stacks is not dictated by the architecture. Instead, it should be based on the requirements of the application.

Stack usage for an implementation of this architecture falls into one of two classes. First is that of single frame usage. In this class, it is used as a traditional LIFO stack, which requires only a small amount of space. Extensive tests have been performed to determine the optimal size of a Data Stack [5, §6.4.1]. These tests show that a stack size of 32 elements is sufficient for all but the most deeply recursive programs.
The second class of usage is with multiple, simultaneous stack frames on-chip (it is also possible to have a single small stack and swap its contents to an off-chip memory). The stack should be large enough to support the required number of frames. The external program controller (or a program running from the instruction stack) must keep a set of stack pointers to reference the frames. These may either be stored externally or on the DS, so a master pointer space may also be required.

The size of the Instruction Stack is entirely dependent upon what applications will be run and how the stack will be used by them. If the entire program is to be loaded, the stack may need to be large; 32 kilobytes has been found to be a very reasonable size for programs encoded in eight bits [5, §4.1.1] when run from external memories. Alternately, the IS could be used to store often repeated subroutines to gain a performance boost. Finally, it may only need to execute a small, very simple operation (as in a systolic array application), requiring a minimal stack size.

Another issue affecting the size of both stacks is the word size selected for the implementation, since this determines the maximum number of elements on the stack which can be directly addressed. Instructions must be provided to access elements on the stack other than TOS, since only being able to access the first element limits the complexity a program is able to achieve. Just as with a machine
organized around a memory subsystem, the amount of memory which can be accessed is limited by the width of the address storage and transport system, which is the stack in this architecture.

The word width of the IS is also related to the word-size issue. The DS must support the full word width to handle data, but the IS only needs to hold instructions, which are only 8-bits wide. The only cases where a wider IS is necessary are when the IS will be used to store data or constants inside a code module for repeated use during execution, or when the implementation uses an extended instruction format to support instructions outside the definition. To handle all cases, it is recommended that the IS be word-width.

2.4.2 Interrupts

Many applications require the ability to interrupt processor activity to perform special, usually time-sensitive operations. This is particularly important when external, asynchronous devices are involved which are running slower than the internal speed of the processor; it is more efficient to continue normal operations than to wait for the device to become ready.

Normally, interrupt handling is a matter of stopping the normal instruction fetch cycle, saving the state of the processor (so the interrupting process has complete
use of the processor, and the interrupted process can be completely restored afterward), and beginning execution of the interrupt handling code. With out a standard instruction fetch cycle, IXA requires a different approach.

Interrupts are handled through a priority-based request system. When an interrupt is signalled, the processor stops it current activity and signals to all external program controllers that an interrupt has occurred at a particular level. Four interrupt levels are directly supported; three are maskable using an interrupt mask code stored in the processor’s status register. An interrupt request succeeds if its level is greater-than or equal to the mask level. This means that level 0 is non-maskable.

Once the interrupt is accepted, the processor pushes the status word onto the DS. This is the only item pushed as a matter of maintaining high-performance and simplicity. If the interrupting process modifies any other system variables such as the stack pointers, IO Mode information, or either of the stacks, it is responsible for properly restoring them, if necessary. The basic restoration process is to clear the interrupt and then restore that status word.

The current interrupt level is communicated to external processors through a set of dedicated signal lines; the information is not stored in the status word. As soon as the interrupt is accepted, external devices should stop sending information
to the processor until the interrupt condition is cleared. This prevents data loss and gives the interrupting device complete access to the processor.

When the interrupting device is through with the processor, it should restore the state and clear the interrupt status. Note that state restoration is left to the interrupting device. This is because an interrupt is handled more as an indefinite transfer of control to another device than one which has some expectation of ending.

An interrupted process can itself be interrupted. The same process as above is performed in such a case. Note that care needs to be taken not to clear the interrupt status under these circumstances as this will preclude proper nesting of interrupts. Instead, a process interrupting another interrupt process should first determine whether an interrupt condition already exists, and, if it does, either wait for the interrupt to complete, or not clear the interrupt status upon exiting its routine.

Interrupt handling at the processor may not be as efficient as handling it at the program controller when the processor is not executing from the stack, since it is the program controller which is sending the instruction stream to the processor and needs to be interrupted. Even if the processor is executing instructions from
the stack it is possible to handle interrupts at the PC since instructions arriving at
the ports have a higher priority than those coming from the stack (in other
words, this can be used as another form of interrupt mechanism).

2.4.3 Levels of Implementation
The development of the computer industry over the past ten years has demonstrated
that no one computer system is correct for all applications. However, there are
many instances where one architecture can be applied to a variety of situations
on different scales. To address this issue, the IXA has been divided into three
levels of standard implementation. Each of these supports a subset of the features
defined by the architecture, and, in doing so, a subset of the instruction set.

The first level, Level 0, supports only the most basic set of instructions. It has
only a minimal Data Stack. Without an Instruction Stack, there is no need to
support any of the stack-based execution instructions or modes. Interrupts also
need not be supported, since such an implementation would rely entirely on an
external program controller for direction, so the controller would be required to
deal with interrupts. Applications for such a design would be primarily found in
array processing where the data-passing capabilities of the processor could be
coupled with a shared instruction-feed mechanism, as is found in many Single
Instruction, Multiple Data (SIMD) machines.
A Level 1 system might be applied where some amount of autonomous instruction execution is required, for example in a reconfigurable array in which each processor may perform a different operation.

Level 2 represents the complete architectural definition. All aspects of IXA and its instruction set must be present in such an implementation. A full implementation with large Data and Instruction Stacks could be used as a stand-alone computing system, or as a powerful coprocessor.

The three levels form a layered model of the architecture:

Diagram 2.2 - Architectural Levels

This model does not directly depict the fact that each layer is also extensible, so
an implementation may support either some instructions of the layer above, or some extended instructions unique to the implementation. This can be graphically shown in a similar manner:

![Diagram 2.3 - Architectural Layers of Definition](image)

The three layers shown in Diagram 2.3 apply to each level defined above. The Mandated area represents the instructions and architectural features required for a particular implementation to be compliant with one of the levels. The Defined region contains features not required by a particular level, but defined for higher levels, which can be implemented if necessary (a design may use some features of a higher level without meeting all of the mandated requirements of that level). Finally, the Extensible region covers all features not provided for by the architectural definition at any level; these are features added by the systems developer.
When these two views of the architecture are combined, it becomes clear how flexible it is with respect to implementation. It also shows how much responsibility is placed on the system implementer to integrate the processor into the total system.

The concept of developing an architecture in many levels is not new. A similar approach was used in the design of the Intel 80960 architecture [8, p. 2]. This architecture directly supports three layers representing core, numeric and protected functions. A fourth layer, the extended architecture, is reserved by Intel for its own extensions. Limiting extended features in this manner reduces the processor's appeal as a generic solution to a specific problem.

2.5.0 Instruction Set Design

With the important features of the architecture defined, an instruction set can be developed. Since the architecture is based on that of a stack machine, many of the instructions will naturally be similar to those found in stack-based computers. The major differences are that (1) the I/O interface is significantly different, and (2) the lack of a code fetch mechanism and the use of a stack for the internal execution of code require special consideration.
The basic functional areas which must be addressed for this architecture are:

Arithmetic & Logic - basic logical functions, shifts, and arithmetic operations.

I/O - providing ability to read and write data under programmed control.

Control - handle setting and reading of processor status information.

Program Execution - allow for conditional execution of code, branching.

These operations are similar in many cases to Forth language primitives. The Forth language was developed to provide efficient execution of code based on a stack machine model. The core set of operations performed on stacks can be easily described in terms of Forth operations, which provide a point of reference. Also, since Forth maps directly to a stack machine, it should not be difficult to develop a compiler to translate Forth to this architecture. The references throughout this document to Forth are aimed at showing this inherent level of compatibility. For more information on the Forth language as it relates to stack machines, see [5, §3.3].
2.5.1 Arithmetic & Logic Instructions

The selection of which arithmetic and logic operations to implement is based primarily upon the examples set by many architectures. The Arithmetic & Logic Unit (ALU) does most of the useful work in the processor, and there are several basic operations which are universal. Additionally, some consideration needs to be given to the types of applications the processor may be used for, namely data processing in which logical and shift operations are useful. The following instructions were selected with this in mind:

- **NOT**: bit inversion
- **AND**: bit disjunction
- **OR**: bit conjunction
- **XOR**: bit inequality
- **SLL**: shift left, logical
- **SRL**: shift right, logical
- **SLA**: shift left, arithmetic
- **SRA**: shift right, arithmetic (sign extend)
- **SLR**: shift left, bit rotation
- **SRR**: shift right, bit rotation
- **ADD**: signed addition
- **SUB**: signed subtraction
- **MUL**: signed multiplication
- **DIV**: signed division
A few notes about instruction selection:

a. All arithmetic operations take signed (two's complement) values as operands.

b. XOR was selected as an independent operation even though it can be performed through a combination of lower-order operations because it is useful in performing operations such as the Cyclic Redundancy Check (CRC) found in data communications.

c. The MUL and DIV operations will not return values outside the defined single-word range of the implementation. This means that not all possible inputs generate results in range. This is in keeping with the minimal approach. Requiring a complete answer to be produced requires the ability to handle double-words, which in turn requires more hardware just to support two instructions. Further, since no other instructions handle double-word values, a programmer making use of the entire double-word result will have write additional code to use in with addition or logical operations, or even store or manipulate it on the stack. It seems better, then, if additional code is required anyway, to provide basic multiply and divide operations which can then be extended by the programmer to handle double-word results.
As a stack machine, the ALU will take its operands from the top of stack (TOS) (one or two operands, as appropriate), and return a result to the TOS; the operands are destroyed in the process. In cases where there are two operands, it may be useful to keep one for additional calculations, like a constant function on a hand calculator. In many applications a constant is often applied to an entire data set. To facilitate this, operations which take two operands have two modes of operation: normal and constant mode. To denote constant mode, a ‘C’ will be appended to the standard mnemonic. The operations supporting constants are:

- ANDC
- ORC
- XORC
- ADDC
- SUBC
- MULC
- DIVC

A final operation which can be classified as an arithmetic operation is a comparison operation. Generally, comparison operations are simply subtractions in which no result is stored. In a stack machine, the disposition of the operands is in question. The program may require that both operands be destroyed, that one be saved (as
in constant mode above), or that both be saved so that an additional, conditional operation be performed on them after the comparison. As such, three compare operations are provided:

- **CMP**: compare, destroying both operands
- **CMPC**: compare, destroying only the operand at the TOS
- **CMPN**: compare, preserving all of the operands.

To flag the result of an operation, a standard set of flags is used:

- **N**: (Negative Flag) - result was negative
- **C**: (Carry Flag) - a carry occurred
- **V**: (Overflow Flag) - arithmetic overflow occurred
- **Z**: (Zero Flag) - result was zero

The N flag is set when the most-significant bit (MSB) of the internal word-size (i.e. after the external instruction flag bit is discarded) is set, as it is for all negative two's complement numbers. The C flag is set when an arithmetic operation causes a carry-out to occur, or when a bit which is set is shifted out during a shift operation. The V flag is set on and add or subtract when the last two carries are not the same (indicating the result was out of the representable range), on a divide if the resulting quotient is outside the representable range, or on an arithmetic
left shift if the shifted-out bit is not equal to the resulting MSB. Zero is indicated when all bits of the result are zero. The flags are only set by ALU operations or direct manipulation of the status register; other operations which simply manipulate items on the stack do not affect the previous settings of the flags.

2.5.2 Stack Manipulation Operations

Just as register-based processors have operations to move data to, from and among registers, a stack machine has instructions which move data to, from and within the stack. For this architecture, only a very basic set of operations is provided, from which many standard stack manipulation operations can be derived.

The most basic operation on a stack is to push items onto it, and pop them off. This is handled essentially transparently by the processor, which automatically pushes data items, and pops operands to instructions as necessary. The READ instruction, a programmed method of pushing data, as well as the WRIT instruction for outputting data, are discussed in section 2.5.3.

Another way to remove data from the stack is to adjust the stack pointer such that it points to the second item in the stack, thereby moving the top of stack. Doing this does not actually delete the old item; it simply becomes invisible to
the next operation. In Forth, this operation is DROP (in Forth's virtual machine environment, the item is destroyed). By allowing the stack pointer to be changed by two positions, Forth's DROP2 can be executed. This leads to the development of the ADJ instruction, arguably the most powerful instruction provided by the architecture.

Since there is no defined stack size, how ADJ is used is relative to the implementation's stack size. Using the ADJ instruction, it should be possible to move the stack pointer by any offset, allowing any number of items to be dropped. On a small stack, adjusting can be used to simulate a register-based system by relatively addressing the various stack elements. On a large stack buffer (several hundreds or thousands of bytes), this addressing can be extended to allow multiple simultaneous stack frames, each of which could be accessed by adjusting the stack pointer. The effect of this is to allow multiple processes or separate data storage areas to use the stack memory without having to flush one processes stack frame prior to another's activation, as is common with other stack machines.

The ADJ instruction applied to the IS (using the mnemonic ADJI) provides the capability to perform relative branches or jumps in the executing code. As with the DS, multiple active frames, in this case containing executable code, can be loaded. These frames may contain only subroutines used by one program, or complete processes, depending on the size of the stack and the application.
In standard architectures, branches are usually made conditionally. To facilitate this (as well as allow adjusts made on the DS to occur conditionally), a condition code may be specified with the adjust. See section 2.5.7 for a discussion of conditional operations.

Two other stack manipulation operations provide the ability to move data on the stack. Unlike a register-based machine in which all valid data items are available to an operation, only the items at the TOS are available in a stack machine. The _ADJ_ operations can be used to change the location of the TOS, but not affect the ordering of items relative to each other. A common operation required to do this is the _SWAP_ function which exchanges the top two items on the stack. Note that this operation requires that two items be written to the stack in one processor cycle; no other instruction requires this, and special hardware may be required to implement the instruction. However, _SWAP_ was found to be among the ten most-common instructions in Forth programs [5, §6.3.1].

Another common instruction in Forth is the _DUP_ operation which makes a copy of the TOS and then pushes it. This is generally used to maintain a data item which would be destroyed by a later operation. This functionality is partly provided by the constant mode provided for arithmetic operations, but not for other instructions taking operands from the DS. Related to _DUP_ are the _SECOND, THIRD_ and successive instructions which take their respective stack elements and push
new copies onto the stack. The generic form of these instructions is the PICK instruction which takes an operand naming a particular element on the stack to copy.

To provide this capability, the architecture has the GET instruction. The basic GET instruction takes an offset encoded within the instruction from the TOS, allowing one of the top 16 stack elements to be copied to the TOS with a single instruction. LGET takes the offset as an operand from the DS, allowing access to $2^{\text{word}}$ elements. GET 0 provides the DUP operation, as does LGET 0 (for LGET, the offset is calculated after the operand containing the offset information is popped from the stack so that GET and LGET operate interchangeably for the top 16 stack elements). Note that by using the GET operations, operands can be fetched from anywhere in the stack to the TOS, and, after a non-constant mode operation is executed on them, the only change to the stack is the result of the operation which is at the TOS. This allows several operations against a data set stored on the stack to be performed (albeit inefficiently).

Two additional instructions which operate on the stacks are PUSH and PUSHI. These allow data from one stack to be transferred to the other. PUSH pops the next item on the IS and pushes onto the DS. PUSHI performs the reverse operation. PUSH is useful for allowing constants to be placed inside a code module that will be loaded into the IS so that they may be pushed onto the DS at the appropriate
time (or multiple times) during program execution. \texttt{PUSHI} allows the IS to be used for storage of results (which can be recovered with \texttt{PUSH} or written with \texttt{WRITI}). Note that executing \texttt{PUSHI} from the instruction stack has the possibly deleterious effect of causing the \texttt{PUSHI} instruction on the IS to be overwritten with the pushed data, which will then be executed as the next instruction. The instruction's ability to be executed under these conditions is of questionable value.

2.5.3 Input/Output Instructions

In addition to the implicit push operation performed when data arrives at a port, it is also useful to be able to read data from a port under programmed control, particularly when the default prioritization of the data does not meet the program's requirements. To allow this, there is a \texttt{READ} instruction which allows a single data item to be read from a specified port and stored on the DS. A \texttt{READ} will cause the processor to halt until data is available at the specified port.

To output data, the \texttt{WRIT} instructions are provided. The basic instruction pops the top element of the DS and sends it to the named port. As with arithmetic instructions, a constant mode of operation is provided (\texttt{WRITC}) so that only a copy of the data item is written; the original remains on the stack. Finally, the \texttt{WRITI} instruction is used to write the next element in the IS to a port. Note that there is no constant mode of operation available for the IS, since to execute such
an operation would cause the written data item then to be executed as an instruction. The purpose of the instruction is to provide the ability to encode constant data as an instruction (so that it can be read directly to the IS) which can then be written directly to a port during IS-based execution.

2.5.4 Control Operations

Control operations provide an interface between the state of the processor and the program executing on the processor by allowing that state to be read and modified directly. Several registers are provided to do this. There are two stack pointer registers: one for the Data Stack Pointer (DSP), and one for the Instruction Stack Pointer (ISP). These registers are at most word-sized, but may be smaller as required by the implementation. The Input/Output Control Register (IOCR) has at most eight bits. The even-numbered bits (0,2,4,6) represent the setting of the word-size-plus-one bit of the output ports, normally the instruction bit. It may be used to pass instructions to other processors or as a separate signalling line. The odd-numbered bits represent the lock-out status of each port; the bit is set for the port if the port is locked (meaning no input is possible on the port, although output is still possible). Finally, the Status Register (SR) contains the processor's Status Word (SW). The eight bits in this register represent the following (from LSB to MSB): Zero flag (Z), Overflow flag (V), Carry flag (C), Negative flag (N), Interrupt Mask bits 0 and 1 (M0, M1), and the internal status bits 0 and 1 (S0, S1).
Access to all of these registers is provided through a group of control instructions. When a register is read, its contents are pushed onto the DS. The instructions to do this are:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSW</td>
<td>Push Status Word Register</td>
</tr>
<tr>
<td>PIO</td>
<td>Push I/O Control Register</td>
</tr>
<tr>
<td>PSP</td>
<td>Push Stack Pointer (DS)</td>
</tr>
<tr>
<td>PSPI</td>
<td>Push Stack Pointer (IS)</td>
</tr>
</tbody>
</table>

When PSP is executed, the value of the stack pointer which gets pushed is the value prior to the push operation (since the DSP is decremented when the value it pushed). If PSPI is executed from the IS, then the value of the ISP which is pushed is that after the PSPI instruction has been popped.

Once a value has been pushed, it may be modified and then used to set the register (or a new value may be loaded and used to set the register). The instructions to set the control registers are:
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSW</td>
<td>Set Status Word Register</td>
</tr>
<tr>
<td>SIO</td>
<td>Set I/O Control Register</td>
</tr>
<tr>
<td>SSP</td>
<td>Set Data Stack Pointer</td>
</tr>
<tr>
<td>SSPI</td>
<td>Set Instruction Stack Pointer</td>
</tr>
</tbody>
</table>

SSP and SSPI are different from the other register-setting instructions in that they can be executed conditionally. This is because, like ADJ, they can be used to modify program flow when applied to the IS.

Another control instruction which does not access any user-register is Clear Interrupt (CLRINT), which sets the interrupt status to normal. The copy of the status word pushed onto the DS when the interrupt occurred is not restored by this instruction; this must be done programmatically. Also, since this instruction simply clears the interrupt status, an interrupt process which was itself interrupted is not restored properly if the interrupting process executes CLRINT. If this is a problem, then either the SSW instruction should be used to set the interrupt mask to lock out further interrupts, or a semaphore system should be established.
To load, enter and exit programs executing internally (using the IS), several instructions are available. They are:

- **LOAD**: Enter loading mode; pass incoming instructions to the IS
- **RESUME**: Return to normal mode from loading mode
- **EXEC**: Begin stack-based execution
- **HALT**: Stop stack-based execution

**LOAD** places the processor in loading mode so that incoming data tagged as instructions are pushed onto the IS instead of being executed. This continues until the **RESUME** instruction is encountered, which causes loading to cease. Note that the **RESUME** instruction is not pushed onto the IS, and, further, it would never be executed from the IS even if it were pushed there using **PUSHI**. **LOAD** can be executed from the IS causing a mode shift from executing to loading which would be terminated by an incoming **RESUME**.

**EXEC** causes instruction execution from the IS to begin. This can be executed conditionally (allowing for conditional subroutine calls, assuming the subroutine
is stored on the IS). Note that an **EXEC** can be simulated by using the **SSW** instruction to change the processor mode to executing (or to loading or normal, for that matter). This would not be conditional and would modify the DS. Stack-based execution continues until the **HALT** instruction is successfully executed. This is also conditional, allowing conditional exit from a subroutine.

Two further program-execution commands, **NOP** and **TTRA** are defined. **NOP**, or No Operation, does nothing except require one cycle to execute. This may be useful for providing delays when I/O between the processor and another processor or program controller is slower than the processor's speed. **TTRA**, or Toggle Trace Mode, toggles the processor's trace mode bit. When this bit is set and the processor is in executing mode, execution stops after each instruction (the mode is forced back to normal mode, a change which is visible in the status word). Execution may be resumed with the **EXEC** instruction. This allows for program tracing during debugging. It could also be used to allow explicit control or synchronization with an external controller during stack-based execution.

### 2.5.6 Extended Instructions

Room is left in the basic instruction encoding for additional instructions to be defined (see section 2.5.7). To eliminate the inherent limitation on the number of instructions that can be defined based on the default instruction size, an escape
instruction, \texttt{EXTOP}, has been defined. This causes the entire next word to be interpreted as an instruction. \texttt{EXTOP} takes its operand from the IS, so that the next word is either the next word tagged as an instruction arriving at an I/O port, or the next word on the IS during stack-based execution.

An implementation may choose not to use \texttt{EXTOP}, but might instead use a wider instruction field for all instructions. This allows extended instructions to use the additional codes provided by the wider word size.

2.5.7 Instruction Encoding

With the instructions defined, it is possible to encode them. There are two goals to be met with the encoding. First, the instructions should be kept as small as possible. A single-word instruction can be loaded in one cycle, whereas multi-word instructions require more time to load and more space to store. The second goal is to also provide as much pre-decoded information as possible in the instruction format to reduce the amount of hardware required to decode the instruction.

To meet the first goal, a word size must be chosen. It may be argued that eight bits is the smallest useful word size, as it is commonly found in analog/digital applications (although a larger word size may need to be supported internally to effectively work with such data). Sixteen-bit operations can be simulated fairly
easily with eight-bit words, and eight-bit words can be conveniently stored in memory (in the case of this architecture, although a ninth bit is required to mark instructions, this is not a problem since 9-bit memory modules are common for use in parity-checking systems).

Using an 8-bit word for an instruction allows a possible 256 instructions to be encoded. The architecture only calls for 54 instructions (plus a certain number to be set aside for extended operations). Since there are extra opcodes available, it is possible to provide some horizontal encoding, that is, some pre-decoded information in the instruction field.

After some initial experimentation with encoding schemes, an encoding method was selected. It calls for the first two or three bits of the instruction word (not including the instruction bit) to be used to determine the group to which the instruction belongs. These group prefixes are:

- **00xxxxx**: Extensible Architecture Group
- **010xxxxx**: Arithmetic Group
- **011xxxxx**: Control Group
- **10xxxxx**: Stack & IO Group
- **11xxxxx**: Stack Pointer Adjust Group
By reserving the 00- prefix for extended instructions, a total of 64 opcodes are left for instructions in the same compact format as the rest of the instruction set (the first opcode, 00000000, is reserved as an escape to indicate that the entire next word is an extended instruction).

The Arithmetic and Control Groups (see Table 2.1) are the most straight-forward in that they are fully encoded with no variable subfields. The 10- prefix is used with nine instructions which generally have some variable information encoded into the lower bits of the instruction (the exceptions being the PUSH, PUSHI and SWAP instructions which are fully encoded). One of the 10- group instructions is GET, which has a four-bit operand field to allow single-instruction access to 16 words on the stack; LGET is provided to access any element of the stack.

The 11- group instructions all contain a three-bit condition field (as do the HALT and EXEC instructions in the 011- group). As discussed above, these instructions can be executed conditionally as a means of providing control over program execution. Only three bits could be provided for encoding conditions, so it was important to decide which conditions should be allowed. The starting point for the selection process was the condition set provided by the Motorola MC68000 family of microprocessors [6, Table A-2]. For that architecture (a CISC architecture), sixteen conditions are available. To determine the static frequency of usage of these conditions in actual applications, code resources in several Macintosh
applications were lexically scanned for branch statements and the results compiled.

The abbreviated results are given below (complete results are listed in Appendix B):

<table>
<thead>
<tr>
<th>Condition (operation)</th>
<th>Total Found</th>
<th>Percent of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any (BRA)</td>
<td>257</td>
<td>31</td>
</tr>
<tr>
<td>Equal (BEQ)</td>
<td>205</td>
<td>25</td>
</tr>
<tr>
<td>Not Equal (BNE)</td>
<td>178</td>
<td>21.7</td>
</tr>
<tr>
<td>Greater or Equal (BGE)</td>
<td>37</td>
<td>4.5</td>
</tr>
<tr>
<td>Less or Equal (BLE)</td>
<td>37</td>
<td>4.5</td>
</tr>
<tr>
<td>Less Than (BLT)</td>
<td>28</td>
<td>3.4</td>
</tr>
<tr>
<td>Greater Than (BGT)</td>
<td>23</td>
<td>2.8</td>
</tr>
<tr>
<td>Plus (BPL)</td>
<td>16</td>
<td>1.9</td>
</tr>
<tr>
<td>Carry Set (BCS)</td>
<td>10</td>
<td>1.2</td>
</tr>
<tr>
<td>Minus (BMI)</td>
<td>10</td>
<td>1.2</td>
</tr>
<tr>
<td>Carry Clear (BCC)</td>
<td>6</td>
<td>0.73</td>
</tr>
<tr>
<td>High (BHI)</td>
<td>5</td>
<td>0.6</td>
</tr>
<tr>
<td>Overflow Set (BVS)</td>
<td>5</td>
<td>0.6</td>
</tr>
<tr>
<td>Low or Same (BLS)</td>
<td>3</td>
<td>0.37</td>
</tr>
<tr>
<td>Overflow Clear (BVC)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>820</td>
<td>100%</td>
</tr>
</tbody>
</table>
Based on these results, as well as a need to provide relatively complete coverage of possible situations (in other words, it was necessary to provide some capability to test certain flags, even though usage of such tests is rare), the following condition codes were selected:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Mnemonic</th>
<th>Boolean</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>ANY</td>
<td>1</td>
<td>000</td>
</tr>
<tr>
<td>Equal</td>
<td>EQ</td>
<td>Z</td>
<td>001</td>
</tr>
<tr>
<td>Not Equal</td>
<td>NE</td>
<td>Z</td>
<td>010</td>
</tr>
<tr>
<td>Negative</td>
<td>NEG</td>
<td>N</td>
<td>011</td>
</tr>
<tr>
<td>Carry Set</td>
<td>CY</td>
<td>C</td>
<td>100</td>
</tr>
<tr>
<td>Overflow Set</td>
<td>OV</td>
<td>V</td>
<td>101</td>
</tr>
<tr>
<td>Greater Than or Equal</td>
<td>GE</td>
<td>N•V + N•V</td>
<td>110</td>
</tr>
<tr>
<td>Less Than or Equal</td>
<td>LE</td>
<td>Z + N•V + N•V</td>
<td>111</td>
</tr>
</tbody>
</table>

Table 2.1 provides a complete listing of the instruction set:
### Table 2.1 - Instruction Set

<table>
<thead>
<tr>
<th>Extensible Arch. Group</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Next Word</th>
<th>Instruction Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EXTOP</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Arithmetic Group

<table>
<thead>
<tr>
<th>Logical Subgroup</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Operation</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOT</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AND</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ANDC</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CR</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OPC</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>XCR</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Shifter Subgroup

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SLL</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>SRL</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>SLA</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>SRA</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>SLR</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>SRR</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

#### Operations Subgroup

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADD</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>ADDC</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>SUB</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>SUBC</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>MUL</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>MULC</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>DIV</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>DIVC</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

#### Compare Subgroup

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMP</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>CMPC</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>CMPN</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

#### Control Group (Direct)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PSW</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>SSW</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>DS</td>
<td></td>
</tr>
<tr>
<td><strong>PSP</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>PSPi</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>PIO</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>SIO</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>DS</td>
</tr>
<tr>
<td><strong>LOAD</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>RESUME</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>NOP</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>TTRA</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>CLRTNT</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Control (Conditional)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>cp</td>
<td>condition</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----------</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HALT</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>condition</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXEC</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>condition</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack &amp; IO Group</td>
<td>1</td>
<td>0</td>
<td>subgroup</td>
<td>operation</td>
<td>port</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IO Subgroup</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>operation</td>
<td>port</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>port</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRIT</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>port</td>
<td>DS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITC</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>port</td>
<td>DS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITI</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>port</td>
<td>IS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct Stack Subgroup</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>operation</td>
<td>port</td>
<td>IS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>IS</td>
<td></td>
</tr>
<tr>
<td>PUSHER</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>SWAP</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Indirect Stack Subgroup</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>operation</td>
<td>offset</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>offset</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LGET</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>Stack Ptr. Adjust Grp</td>
<td>1</td>
<td>1</td>
<td>operation</td>
<td>condition</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relative Adj. Subgroup</td>
<td>1</td>
<td>1</td>
<td>md</td>
<td>sk</td>
<td>cp</td>
<td>condition</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADJ</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>off</td>
<td>condition</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LADJ</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>condition</td>
<td>DS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADJI</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>off</td>
<td>condition</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LADJI</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>condition</td>
<td>DS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute Adj. Subgroup</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>condition</td>
<td>DS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSP</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>condition</td>
<td>DS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSPI</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>condition</td>
<td>DS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1 (continued) - Instruction Set

An Extensible, Scalable Microprocessor Architecture
2.6.0 Instruction Mandates for Each Implementation Level

The three implementation levels discussed above each have a set of mandated instructions from the complete instruction set definition which are required to make an implementation compliant with a particular level. Note that instructions which are implemented beyond these mandates are part of the Defined features set for a level (except for Level 2, where all instructions are required so that any further instructions supported by the implementation are Extended).

The basic criterion for omission of an instruction from Level 0 is whether or not it can be emulated with any other instructions in a reasonable way. To some extent, implementation difficulty was also evaluated, based on experience in developing the two demonstration systems discussed later in this document.

The instructions required for a Level 0 design are:

- **EXTOP** (if used)
- **NOT**
- **AND, ANDC**
- **OR, ORC**
- **XOR, XORC**
- **SLL, SRL, SLA, SRA, SLR, SRR**
- **ADD, ADDC**
- **PSW, SSW**
- **PIO, SIO**
- **PSP**
- **NOP**
- **READ**
- **WRIT, WRITC**
- **PUSH**
- **GET**
- **SWAP**
- **ADJ, LADJ (if stack size warrants)**
- **SSP**
Several of these instructions are only required if the size of the stack makes it appropriate to implement them. For example, \texttt{LGET} is only necessary if the stack is outside the range of \texttt{GET} (sixteen elements). Similarly, \texttt{LADJ} is required if a point in the stack cannot be reached with two \texttt{ADJ} instructions (since \texttt{ADJ} will, at most, require the equivalent of two instruction cycles due to the need to preload the operand).

Level 1 builds on Level 0 by adding an Instruction Stack and the instructions necessary to use it. Again, instructions which are dependent upon the size of the stacks are implemented as appropriate. Also, instructions which can be emulated are omitted. The list of required instructions for Level 1 (beyond those for Level 0) is as follows:

\begin{verbatim}
LOAD RESUME EXEC HALT PSPI
TTRA WRITI PUSI ADJI, LADJI (if stack size warrants) SSPI
\end{verbatim}
3 • Implementation & Testing

3.0.0 Overview

Since the architecture developed above is so flexible in terms of the configuration of the actual processor implementation, choosing a particular implementation limits what types of tests can be performed on the processor. If a minimal design is chosen, then more complex features of the architecture are unavailable for evaluation. Alternately, if a complex implementation is done, the architecture’s performance in minimal systems can not be judged.

In an attempt to demonstrate the architecture in a number of environments, two implementations have been performed as part of this work. The first is a VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) model of a two-port, Level 2 implementation of the architecture. This model is a behavioral model, meaning that only the functionality of the processor is addressed, not how that functionality is internally achieved (i.e. a black box model).

The second implementation is a logic design targeted at a Field Programmable Gate Array (FPGA). This implementation is minimal, providing 1 port and Level 0 compatibility with the architectural definition. The purpose of this implementation is to demonstrate a physical fabrication, as well as show that there are many target technologies capable of supporting the architecture.
3.1.0 VHDL Model

The purpose of creating a VHDL model is two-fold. First, it provides a platform for simulation, and thereby verification of the architecture. Although the architecture is completely defined on paper, without a verification model, there is no way to know whether the given functionality is enough or too much to solve actual problems; the VHDL model provides a means to test this. Second, it provides coded documentation of the architecture’s behavior. Initially, the effect of this was to help clarify certain issues in the architecture which were under-defined in the specification. In its completed state, the code provides a working, tested definition of the architecture for future reference.

As a behavioral model, little thought was giving to accurately simulating the exact means by which an instruction is executed, or the timing issues associated with a particular implementation. Instead, the model was designed to be implementation-technology independent. This allowed for a more accurate test of the architecture separate from any special capabilities or problems associated with a particular implementation technology. As such, any timings or operations seen in the code are entirely arbitrary.
The code was written, compiled and tested under Mentor Graphics version 8.2 design tools, including the Design Architect, VHDL Editor, System-1076 VHDL Compiler, and QuickSim II logic simulator/debugger. A complete listing of the source code is provided in Appendix C.

3.1.1 Entity Definition

The entity definition at the top of the code defines the processor’s interface. This definition provides a black-box style interface in that several different architectural modeling styles could use the same interface, and the compiled object could be integrated with other components which are not necessarily compiled VHDL objects themselves. Diagram 3.1 shows this interface and other system components.

![Diagram 3.1 - VHDL Model Design](image)
The model has two logical ports, each of which is physically divided into input and output ports. Two ports is enough to demonstrate the handling and prioritization of simultaneous inputs. The model could easily be extended to allow more ports (up to the maximum of four supported by the architecture) for testing applications requiring more than two ports. Each physical port has its associated input and output handshake signals.

The ports have been defined as type INTEGER for several reasons. Internally, the model emulates an 8-bit machine. For this, a ninth bit is required at the I/O ports. By using integers (16-bits), no special types need to be defined. Also, since the model is designed to be used in test bench not only for the architecture, but also for the software that will be executed on that hardware, using integers makes it much easier to define an external object that can be used to read instructions and data from a file and pass them to the processor model for simulation.

Also defined are two interrupt request and acknowledge signals, the status word output (also an integer), and the processor reset signal.

3.1.2 Model Internals

The architecture block of the model contains several processes. The two ports are modeled as separate processes which interact with the rest of the processor through
a series of handshake signals in much the same way as is done in the FPGA implementation (see section 3.2.5).

The main body of the architecture is the execution process which is called when data becomes available at either of the ports. This data is decoded and then acted upon either as an instruction or pushed onto the stack. In terms of simultaneous data handling, this model supports the ability to push two data words onto the stack in one cycle, or to execute an instruction on an incoming data word in one cycle.

Internally, most of the process of decoding the instruction is done mathematically; that is, there is no conversion to bit vectors to perform instruction decoding. Conversions to and from bit vectors are used to handle boolean operations since VHDL defines these operations on bit vectors. The setting of status flags by arithmetic and logic operations is handled mathematically using closed-form expressions.

3.1.3 VHDL Test Bench

To test the VHDL model, a second VHDL object was created which would simulate an external program controller. To define how this system would provide an interface to a memory, as well as demonstrate the extensibility of the architecture, a Standard Memory Interface (SMI) extension to the architecture was created.
Note that this extension was not central to the thesis topic, but rather was a tool for testing and experimentation. As a result, it is by no means complete or optimal. See Section 4.4.0 for a further discussion of some issues which need to be researched about such interfaces.

The interface module reads a program from a file; the file is a series of ASCII base-ten integers representing instructions and data. Upon receiving the LOAD signal, the program controller (PC) reads the file from disk into a simulated memory, and then begins execution. Words are read from memory and tested to see whether they are SMI instructions or simply instructions and data for the processor. SMI instructions are executed in the PC, while everything else is passed to the processor. The entire program can be monitored during simulation to check for proper operation.

The Level 0 test program was executed on the model (this is listed in Appendix D), and the expected results were achieved (provided in Appendix E). A very simple program to execute off the stack is also listed in this Appendix. As discussed in section 4.4.0, it should be a goal of future research to use such a model to further test different types of algorithms on the architecture.
3.2.0 FPGA Implementation

Whereas a VHDL model can be used to demonstrate and test the architecture’s capabilities, a physical implementation demonstrates that it can be constructed. This architecture leaves much of the physical implementation undefined, allowing great flexibility. To demonstrate this flexibility, a very basic (Level 0) version of the architecture has been implemented using Field Programmable Gate Array (FPGA) technology. The simplest possible implementation was chosen to provide a benchmark for the lower limit of the architecture’s specifications and to see whether or not this limit provides enough power to create a useful processor.

The purpose behind this approach is to show that inexpensive technology can be used to implement a processor conforming to the architecture. Using the FPGA, not only is the development cycle manageable (this processor was designed, fabricated and tested within a three-month period by the author), but the resulting device can be rapidly reproduced. Additionally, since FPGA development software is fairly flexible, the design can be extended in the future, or used as a “core” for larger systems.

3.2.1 FPGA Technology

Actel FPGA components and development tools were used in the production of the processor [11]. The Actel 1020A FPGA used provides 547 logic modules,
which are simply four-input, one-output logic blocks which can be programmed into a variety of logic configurations through a set of burn-once fuses. These blocks can then be connected together using an internal interconnect network which also uses fuses to permanently set the logic design into the chip. This permanent-burn system tends to allow for simpler FPGA’s, which are therefore cheaper and provide more logic blocks. Additionally, they are faster than reprogrammable FPGA’s (such as FPGA’s manufactured by Xilinx) which rely on static gates to configure the chip.

The schematic capture for the processor was done initially on a Macintosh using LogicWorks 2.5.1, and then transferred to the Apollo systems using Mentor Graphic’s NETED v7 tool. The resulting netlist was then used by Actel’s Action Logic System (ALS) version 2.11 tools to automatically place and route the chip. The resulting file was used to program the chip, which could then be used immediately.

3.2.2 The Processor Design

The implementation done here is compliant with the architecture’s Level 0 specifications. Specifically, it has one logical port (physically separated into input and output ports), a four-element data stack, and an 8-bit data path. This processor
was designated IXAip108 (the '1' refers to the single port, the '08' is the size of a data word). A single letter is postfixed to this to indicate design revisions.

The limitations placed on the implementation by the FPGA technology drove many aspects of the design, most notably the minimalistic approach. Each logic block in the array can support up to four inputs and one output, although not all four-input logic functions can be generated with a single block. Actel's logic chips also require that only bi-state logic be used; this rules out the use of tri-state buses internally, so that multiplexers must be used. These are expensive in terms of logic blocks since at least eight blocks are required for each multiplexer on the bus. The same argument can be made about every element of the data path, so that increasing the data path size proportionally increases the number of blocks for the processor.

The number of I/O ports was limited mainly to make the processor simpler to route (since it appeared that it was going to be densely populated already). An additional port would have required even more logic to support not only the port hardware, but the architecture's capability to handle multiple data items simultaneously.

Speed was also not an issue in the design. The FPGA is not the fastest technology available and would not be the technology of choice for performance-critical
applications. However, the Actel FPGA does offer enough performance that 20 to 30 MHz clock speeds are not uncommon for some applications. The goal of this implementation was simply to break the one million instructions per second (MIPS) barrier, a reasonable performance expectation for such an implementation.

The remainder of this section details the logic of the processor and its components.

3.2.3 Overall Design

The processor is constructed of five major subsystems: the Arithmetic and Logic Unit (ALU), the Input/Output Unit (IOU), the Data Stack (DS), the Control Registers Block (CRB), and the Control Unit (CU). Each of these systems is detailed below. The overall arrangement of these pieces in the processor is shown in diagram 3.2. Also shown are the main internal buses, as well as the primary external interface connections.
3.2.4 Arithmetic/Logic Unit (ALU)

Nearly half of the instructions provided by a Level 0 implementation are arithmetic or logic operations. The ALU handles 8-bit, signed integers and provides three classes of operations: boolean, shifts, and arithmetic. To do this, the ALU has been broken into two sections: the arithmetic/logical section, and the shifter. Data and the current operation are applied to both sections simultaneously, with the result being selected by an output multiplexer.

The arithmetic/logic section consists of eight arithmetic/logic bit-operation modules, each of which performs all operations upon the input data. The appropriate result is selected by a 4-to-1 multiplexer at the output of each module. With the exception of addition, each module operates independently. For addition,
a carry-in and carry-out bit are used. Combining modules in this way creates a ripple-through adder. Although this is not the most efficient design (a carry-look-ahead design would be somewhat faster), it is more conservative in terms of logic module requirements, and the performance loss in an 8-bit system is minimal. The logic for a single module is shown in Diagram 3.3.

The shifter is an array of AND gates which, given a shift direction (encoded in bit 0 of the instruction), selects the appropriate neighboring bit to transfer to the output. The shifter can be set for three modes of operation: logical shift, arithmetic shift, and rotate. In logical shifting, the bits are simply shifted in the selected direction; a set bit shifted out in either direction causes a the carry flag (C) to become set, and zeros are always shifted in to fill the emptied bit position. In arithmetic shifting, the sign of the operand is preserved. For an arithmetic right

Diagram 3.3 - Arithmetic/Logic Bit Slice
shift (SRA), this means setting the MSB of the result if the MSB of the operand was set. For an arithmetic left shift (SLA), the sign is preserved by the nature of the operation, unless an overflow occurs, which is reported with the overflow flag (V). For the rotate operation, the bit shifted out of one side of the shifter is shifted in the opposite side.

The operation to be performed is fed directly to the ALU from the CU in the form of the lower five bits of the instruction; all decoding is performed within the ALU. The decoding logic inside the ALU generates several control signals; these are listed along with their generation logic in Table 3.1.

<table>
<thead>
<tr>
<th>Function</th>
<th>Signal</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invert A</td>
<td>InvA</td>
<td>(11 XNOR I4) AND (NOT I2)</td>
</tr>
<tr>
<td>Enable B</td>
<td>EnaB</td>
<td>I1 OR I2 OR I4</td>
</tr>
<tr>
<td>Enable Shifter</td>
<td>ShEna</td>
<td>I3 AND (NOT I4)</td>
</tr>
<tr>
<td>Enable Arithmetic/Logic</td>
<td>A/LEna</td>
<td>NOT (I3 AND (NOT I4))</td>
</tr>
<tr>
<td>Carry In</td>
<td>Ci</td>
<td>I1 AND I4</td>
</tr>
<tr>
<td>Function Select 0</td>
<td>S0</td>
<td>I1 AND (NOT I4)</td>
</tr>
<tr>
<td>Function Select 1</td>
<td>S1</td>
<td>I2 AND (NOT I4)</td>
</tr>
</tbody>
</table>

Table 3.1 - ALU Internal Control Signals

The ALU is a combinational circuit; it processes the inputs as they become available, and the time it takes to produce a stable result is the main factor determining the maximum speed of the processor.
3.2.5 Input/Output Unit (IOU)

To provide the asynchronous interface required by the architecture, the IOU uses input and output buffers to latch data and provide handshaking between the processor and external devices.

The input side of the IOU waits for the input handshake signal (IHi) to latch data. It will only be able to latch the data if the Lockout (LO) bit for Port 0 is cleared in the Input/Output Control Register (IOCR). Note that setting this bit effectively stops the processor until a reset is performed since there is only one port and therefore only one way to feed instructions to the processor. When the data is latched, the IOU passes the data to the I bus and sets the Data Available (Dav) signal to the CU. When the CU accepts the data, it responds with the Data Acknowledge (Dack) signal, which causes the IOU to drop its Dav signal, and set the output handshake (IHo) signal. When the external device drops its input handshake, the IOU drops the output handshake. Diagram 3.4 shows the timing of this sequence of events.

![Diagram 3.4 - Input Handshake Timing](image_url)
The output side works in a similar manner, but is simpler because the processor internals only need to pass the data out; there is no need to wait for a response from the processor side of the interface as there is when reading data. When the processor executes a WRIT instruction, the CU generates the Outdata signal to latch the data, which is on the Y bus, into the output latch. This also sets the output handshake on the output port (OHo). When the external device receives the data, it signals with the input handshake (OHi), causing the IOU to drop the output handshake. The output handshake is also passed back to the CU to indicate the state of the port; if it is asserted, then the port is still busy and the processor must wait to send further output. Diagram 3.5 shows the output process.

![Diagram 3.5 - Output Handshake Timing](image)

### 3.2.6 Data Stack (DS)

The DS provides storage for operands and results. This implementation has a four-element stack, meaning that four, 8-bit registers are available for data. The stack is addressed by the stack pointer (SP) which is stored in the CRB. Access to the stack is provided by an input bus which feeds the four registers, and a two-level multiplexer array which selects the output.
Normally, the current top-of-stack (TOS) register is selected by the SP0 and SP1 bits and passed to the output. Since it is common to use the second item on the stack for two-operand operations, it can be quickly selected using the SP+1 signal (since the stack grows downward, the item at SP+1 is the second item on the stack), which adds one to the value of the stack pointer inside the DS (without modifying the SP register). This selection mechanism works at both the input and output sides of the DS. On the input side, the DLatch signal is gated to the selected TOS register to latch the data coming into the DS.

In the final design, the processor reset signal was not connected to the registers so that their contents would remain past a reset for examination (in case of an unexpected failure).

3.2.7 Control Register Block (CRB)

The CRB contains three registers which store the processor's state information: the stack pointer (SP), the input/output control register (IOCR), and the status register (SR). All of these provide state information to other parts of the processor, and are modified by other parts of the processor. Additionally, they can be read or written under programmed control. The processor reset signal clears all of these registers.
The SR contains the four ALU status flags (N,C,V,Z), as well as a processor status bit. Since Level 0 does not support an instruction stack, the only valid status settings are 00 (Normal) or 11 (Error), which can be represented with a single bit. Also, since there is no IS, there is no need to support interrupts in the processor (this would be done in the program controller), so the interrupt mask bits are not present. For compatibility, all bits appear in there respective positions when the status word is pushed onto the DS. Inputs to the SR come from either the bus or directly from the ALU. The Error flag is currently not set by the hardware (it would only indicate an illegal instruction), but it can be set in software.

The IOCR contains two bits since there is only one port. Bit 0 is the Instruction Bit for port 0; if it is set, then the ninth bit of port zero is set at the output. This could be used to pass instructions to other processors, or as a separate signal line. Bit 1 is the Lockout Bit for the port; if it is set, the port is locked-out, meaning than no incoming data or instructions will be latched. Although the bit only affects the input side of the port, the fact that there is only one port makes it impossible to perform any output once the bit has been set. Input to the IOCR is only through the bus.

The SP contains the current stack pointer. This 2-bit register has two sets of inputs. The first comes from the bus and is used to load a new stack pointer. The second comes from the CU and is used as an offset from the current stack pointer.
value. A dedicated adder adds the current value to the incoming value and passes it to the DS. This allows operations like GET to access any part of the stack. This offset value can also be independently stored in the register; this is used to perform the ADJ instruction.

All of the registers’ outputs are connected to the bus. Enable lines coming into the CRB select the register to pass to the bus, as well as the register to be affected by the LATCH signal, which stores the data coming into the CRB in the appropriate register.

3.2.8 Control Unit (CU)

The CU consists of an Instruction Register (IR), a clock phase generator, and decoding and control signal generation logic.

When the processor is idle, that is, it is not executing an instruction or loading data, it waits for input at the port, which is signaled by the Dav signal from the IOU. Upon receiving this signal, the execution cycle is initiated by starting the clock phase generator on the next external clock cycle. By not keeping the clock phase generator running (and thereby executing no-operation cycles when there is not data available), the processor is better able to respond to asynchronous input (it only needs to wait until the next clock cycle rather than the next processor
cycle). A complete processor cycle is four clocks long. The phase generator provides eight overlapped phases. Even-numbered phases (0,2,4,6) are setup and hold phases which last for one cycle. During this time, the next event is setup internally (i.e. data is routed and allowed to become stable). The odd phases occur at the midpoint of each even phase and are the active phases. For these, the rising edge of the phase is used to clock data into the next stage of the operation. When the cycle completes, a test is made to determine whether more data is already available. If it is, the next cycle is immediately begun. Otherwise, the processor becomes idle. Diagram 3.6 shows the timing of the clock phases.

<table>
<thead>
<tr>
<th>EXTCLK</th>
<th>PHI0</th>
<th>PHI1</th>
<th>PHI2</th>
<th>PHI3</th>
<th>PHI4</th>
<th>PHI5</th>
<th>PHI6</th>
<th>PHI7</th>
</tr>
</thead>
</table>

Diagram 3.6 - Clock Phase Timing

If an instruction is to be executed (this is indicated by the Inst signal from the IOU), the instruction is latched into the IR phase 1. The control signal generation logic then creates the necessary control signals for the operation through an AND - OR type array. Signals are gated against the appropriate clock phase before distribution to the other components. A complete table of the signals generated and the phases in which they are generated is given in Table 3.2.
<table>
<thead>
<tr>
<th></th>
<th>LatchD0</th>
<th>LatchD1</th>
<th>Pass</th>
<th>SelA/B</th>
<th>SelIn</th>
<th>SelReg</th>
<th>EnaOCR</th>
<th>EnaSR</th>
<th>EnaSP</th>
<th>EnaCC</th>
<th>LatchReg</th>
<th>Dack</th>
<th>OutData</th>
<th>DSLatch</th>
<th>SP+1</th>
<th>LoadSP</th>
<th>SPA1</th>
<th>SPA0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>6</td>
<td>5,7</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>NOT</td>
<td>AND</td>
<td>ANDC</td>
<td>OR</td>
<td>ORC</td>
<td>XOR</td>
<td>XORC</td>
<td>SLL</td>
<td>SRL</td>
<td>SLA</td>
<td>SRA</td>
<td>SLR</td>
<td>SRR</td>
<td>ADD</td>
<td>ADDC</td>
<td>SP+1</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>PSW</td>
<td>SSW</td>
<td>PSP</td>
<td>PIO</td>
<td>SIG</td>
<td>NOP</td>
<td>READ</td>
<td>WRIT</td>
<td>WRITC</td>
<td>GET</td>
<td>SWAP</td>
<td>ADJ</td>
<td>SSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>3,6</td>
<td>3,6</td>
<td>3,6</td>
<td>3,6</td>
<td>3,6</td>
<td>3,6</td>
<td>3,6</td>
<td>3,6</td>
<td>3,6</td>
<td>3,6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2 - Control Signal Generation
The standard processor cycle is as follows:

Phase 0 (1): Latch TOS to D0 Register
Phase 2 (3): Acknowledge data (Dack), latch second operand to D1
Phase 4 (5): Adjust stack pointer as necessary to store result
Phase 6 (7): Store result

All ALU instructions follow this pattern. Because the ALU is a combinational unit, as soon as the second operand is clocked in on phase 3, the computation can begin. The result is routed back to the DS and stored on phase 7, providing two cycles for the calculation.

Control instructions function similarly, except that the second operand is either read from the CRB (any of the push operations), or the first operand is written to the CRB (any of the set operations).

The SWAP instruction uses the SP+1 signal to both obtain its second operand and to store the first. The first operand is stored back to SP+1 on phase 5, while the second is stored at SP on phase 7. This is the only operation in which two results are written in one cycle.
There are several special cases. When there is no instruction, but rather data arriving at the port, a READ cycle is simulated. Unlike a normal READ cycle, phases 2 and 3 are skipped by sending the SKIP2 signal to the phase generator. This has the effect of keeping the Dack signal from becoming asserted and clearing the input register of the IOU. Instead, the processor moves on and latches the data on phase 7, and then asserts Dack. This also allows data loading cycles to be executed more quickly and efficiently.

The READ and WRIT cycles are normally initiated by those instructions, and the Dack signal is sent on phase 2, clearing the input register. In the case of the READ instruction, data needs to be present by phase 7 so that it can be stored in the stack. If data is not available, the processor cycle is frozen in phase 6 by asserting the Hold6 signal to the phase generator. This signal is cleared when Dav is asserted by the IOU. Similarly, when the WRIT instruction is executed, the processor expects the output port to be free on phase 7. If it is not, the Hold6 mechanism is used to wait until the port becomes clear. Note that port will only be busy if data placed there by a previous WRIT instruction was not handshaked out by an external device.
3.2.9 Additional Design Notes

All of the additional top-level logic is used for routing data. Several main multiplexers are used in the data path. Going into the DS is the input multiplexer which selects between the Y bus and the IOU input bus (when SelIn is asserted). The operand multiplexer selects the source of the operands going to the D0 and D1 registers from either the DS or the CRB (when SelReg is asserted). The Y bus multiplexer selects between results coming from the ALU or data coming directly from D0 or D1 (when Pass is asserted). The source of the passed data is normally D1, unless SelAB is asserted, in which case the source is the D0 register.

3.2.10 Design Testing

The design was initially entered in LogicWorks, a Macintosh-based schematic capture/logic simulator. Generic parts were used to verify the logic of the design. This version, IXAip108a, was tested at the module level.

This design was then transferred to the Mentor Graphics version 7 tools and modified to take advantage of the Actel libraries. This process resulted in Revision B of the processor design.

Prior to programming the FPGA, extensive testing of the circuit model was performed using the design tools. Initially, each module was tested by hand in
QuickSim using the default delays to verify the logic design and schematic entry. A few bugs were caught at this point and corrected. The most critical test (and the most complex) was the test for the CU; every instruction was loaded into the instruction register and a complete cycle executed to ensure that the proper signals were being generated at the proper times.

Once the top-level design was completed, each instruction was again tested, this time against data loaded into the stack. No formal test program was run yet, because there was some concern as to whether or not the design was too large to place and route on the chip. A place-and-route was run at this stage, with positive results (approximately 74% utilization).

After routing, it was possible to back-annotate the design file with delays more accurate to the actual layout. A test simulation of the back-annotated file was not particularly encouraging, as the processor locked-up after a few instructions. The problem was traced to the point where the external, asynchronous inputs are synchronized with the external clock used to generate the processor's clock phases. Occasionally, if the incoming data were too close to one edge of the external clock, a setup problem occurred. This was corrected by adding a buffer stage to the cycle timing, the result of which was to make the processor cycle one clock
longer. In a VLSI implementation, this problem could be eliminated by making the system truly asynchronous using an on-chip oscillator triggered by the incoming data.

With the cycle timing reworked, a second place-and-route was performed (leading to Revision C), this time with successful back-annotated simulations. The standard Level 0 test was run, with mostly successful results. One major problem uncovered at this time affected several instructions and was timing-related. In order to reduce the size of the design, transparent latches were used in several places. Certain timing situations caused data sent to these latches to be improperly stored if the data changed prior to the end of the latch signal (note that the latch signals were originally intended to be edge-triggering, not level-sensitive). This problem was solved by replacing those latches with true, edge-triggered registers. Again, a place-and-route was performed and, although the FPGA resources utilization was 10% higher (due to the use of edge-triggering devices), the design was still routable. Further simulations met expectations; results for the Level 0 test program are provided in Appendix G. The final version, IXAip108d was then prepared for programming into the FPGA chip. The schematics used to program the chip are shown in Appendix F.
3.2.11 FPGA Test Bench

To test the FPGA implementation, a simple board with connections to the proper I/O's was built. Since there is no need to add additional memory or support chips, very little wiring and time was required to get the processor working. The pin placement information generated by the development system (see Appendix H) was used to wire the processor socket to the I/O sockets. Diagram 3.7 shows the pin placement in the 68-pin PLCC chip.

![Diagram 3.7 - Pin Placement](image-url)
To provide the necessary signals and output indicators, two standard switch/lamp boards were attached to the processor board through ribbon cables. A basic interconnection diagram is provided below (Diagram 3.8).

\[ \text{Diagram 3.8 - Processor/Switch Board Test Bench} \]

A second test bench was also build to demonstrate the ease with which the processor can be interfaced with other devices. An MC68008 board was connected as shown in Diagram 3.9, and a simple software program (listed in Appendix I) was written to feed instructions to the processor and store results. Little time was spend on the system, although it could be easily expanded to provide a basis for a more powerful program controller and test environment.
The time spent debugging the design with the design tools proved worthwhile when the processor was finally programmed into the FPGA chip; the processor worked 100% without any further modification. The system was tested up to 4 MHz (the limit of the available clock generator) at the high-end, and as slow as hand-clocking would allow at the low end, demonstrating the asynchronous capabilities of the architecture.

3.3.0 General Implementation Notes

The FPGA implementation performed here was simple but provided many insights into designing for the architecture. This section discusses a few of these ideas which could be applied to future designs.
The FPGA’s requirement that multiplexers be used to implement internal buses necessarily limits the size of the memory which can be built into the processor. Additionally, it limits the rate at which the memory can be accessed. It would be much more efficient to acquire both operands from the stack in one cycle instead of two. Also, to support the ability to load multiple data items (and execute an instruction against them) in one processor cycle requires a more efficient internal bus.

Along these same lines, the ALU should be implemented with the faster logic such as a carry-look ahead adder. Since the processor’s other circuitry requirements are minimal (in terms of complexity), a fast multiplier for Level 2 implementations could be used instead of one which performs the operation in less hardware, but more time.

Another problem facing larger implementations is the number of pins necessary to support up to four ports with large word-sizes. This is even more of a problem if the ports are physically separated to improve performance. The requirements for different scales of the architecture are listed in Table 3.3. Note that this table does not optimize for unnecessary status word pins (as was done with the FPGA design), and does not include power and ground pins (which will vary according to the requirements of the design). With larger implementations, trade-offs will probably be made necessary by pin limitations.
<table>
<thead>
<tr>
<th>Word Size</th>
<th>Logical Ports</th>
<th>Physical Ports</th>
<th>Interrupts</th>
<th># of Pins</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>55</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>53</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>91</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>171</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>89</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>159</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>159</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>299</td>
</tr>
</tbody>
</table>

Table 3.3 - Pin Requirements for Various Processor Scales

As mentioned above, the timing cycle may be shortened by combining IXAip108’s two operand fetch cycles into one clock cycle. However, the minimum number of unpipelined cycles is probably three: operand fetch/instruction decode, execute, result store. On-chip clock multiplying technology found in recent microprocessors could be used to generate high internal performance. The completely asynchronous interface demonstrated so clearly with this implementation allows the processor internals to be truly independent from the external systems. It may even be the case that the internal system runs fast enough that it appears to support multiple data items per cycle handling, while it really does not, simplifying the design.

Pipelines and superscalar designs are a common RISC (and CISC) approach to improving performance. As discussed above, both of these are difficult to implement in a stack machine because of the inherent data dependencies. Some pipelining could be accomplished by folding the result storage cycle with the
operand fetch cycle, reducing the required number of cycles per instruction effectively to two (the theoretical minimum for a stack machine [5, §6.2.3.1]). This approach would also not suffer the problems associated with branches, because the instructions themselves would not be pipelined. If the processor implementation supports a floating-point extended instruction set, integer and floating-point instructions could be handled simultaneously, as in superscalar designs.

Another design optimization which should be investigated is a common technique used in procedural-language compilers called combining and deferring stack adjusts. This technique reduces pushing and popping of data from the stack when it is known in advance that the data will be overwritten or used immediately by the next operation (as discussed with pipelining above). At the end of a period of predictable transactions against the stack, the correct results are placed on the stack where they would have normally ended-up, and the stack pointer is properly adjusted. This is already done to some extent by IXAip108 in that the operands to an operation are not really popped off the stack and the result pushed. Instead, the stack pointer is simply adjusted and the data written back to the stack. This only occurs within a single instruction; the suggestion here is that the technique could be applied across several operations.

A final consideration which impacts possible uses for the architecture is power
consumption. Since the processor is completely idle when there is no activity at the ports, it uses almost no power, automatically. No additional hardware is required to monitor power usage as is commonly used in notebook computers to extend battery life. Calculated power consumption for IXAip108 is plotted in Diagram 3.10.

![Diagram 3.10 - Dynamic Power Consumption](image)

This result is based on the formula [12, p. 2-9]:

\[
\text{Power (mW)} = (0.20N \times f1) + (0.80P \times f3)
\]

Where \(N\) is the number of logic modules used, \(f1\) is the average module switching
rate (in MHz), P is the number of I/O's with a 50pF load, and f3 is the I/O average I/O switching rate. Taking N = 460, P = 17 (the outputs), f3 = 1/5 f1 (I/O’s switch once per cycle), the power consumption at the nominal 4MHz speed is estimated at 400mW. This is, of course, only if the processor is kept active. If it is not, then these results can be scaled by the percentage of active cycles.
4 • Discussion

4.0.0 Overview

Part II provided some discussion of IXA relative to other computer architectures as they related to the design process. This part builds on that discussion by comparing many of the major features of IXA with other modern architectures. Possible applications are also discussed. Possible directions for further research and development of the architecture are also given.

4.1.0 IXA & RISC

The RISC philosophy reduces the complexity of a processor (and improves its performance) by applying several design tenets [11, §8.1.2]:

- Use a load/store approach to limit memory access
- Require that all instructions execute in one cycle (optimally). Instructions which exceed this should only be included if they are useful enough to offset the performance loss and increased complexity they require.
- Encode instructions horizontally to reduce decoding.
- Use caches, pipelining and super-scalar designs to improve performance.
- Use large register files for local variable storage and parameter passing.
Memory access is left entirely to the implementation. Instead, a high-speed, basic external interface is used resulting in very high I/O performance. It is not possible to pipeline the instruction execution mechanism except in rare cases when there is no contention for the data at the TOS. By introducing some of the performance optimizations discussed in section 3.3.0, the cycle time can be reduced to two clocks per cycle. Since the processor can function at the limit of the technology internally, this is not much of a limitation.

There is verly little horizontal encoding, mainly since there is no register-based addressing scheme. The width of an instruction is one byte, one-quarter the standard width of a RISC instruction. This leads to far more efficient use of memory and simpler interconnection schemes.

This is not to say that IXA is superior to RISC. On the contrary, IXA is not well suited to operate in the RISC stronghold area of high-performance workstations. This is because RISC is optimized for an operating system environment and the types of applications and hardware found there.
4.2.0 IXA & Stack Machines

Many other stack machine architectures have been done in the past, so IXA is not revolutionary in this respect. However, several elements of the architecture are unique to stack machine design and bear some discussion.

IXA supports true zero-operand addressing. The stack is the implicit destination of data items. Some other stack-based systems allow for more than one stack or several user-registers usually to improve performance. Doing this requires some form of addressing to direct the data to the appropriate location, a departure from one of the stack's strongest features.

The use of a stack for storing and executing instructions is not found in any other design. In IXA, this is a result of the 0-operand load mechanism and a need to boost performance over that which can be obtained by relying solely upon an external program controller to feed instructions and data, and process program branches. A similar idea regarding the use of a stack to hold code (but for safe self-modifying code) is discussed in Koopman [5, §9.5.2].

Conditional subroutine entrance and exit is another feature not often found in other processors. This, combined with the use of an instruction stack, is of particular interest on a stack machine because it allows for zero-cost conditional clauses (such as IF and CASE statements) found in high-level languages. This is also
based on an idea suggested in Koopman [5, §9.5.1] where it is noted that such a structure can be efficiently executed through the use of conditional subroutine returns. In IXA, the body of the IF clause can be stored on the IS and conditionally entered (the initial IF), or exited.

4.2.1 IXA and the Transputer

Of all other architectures, the Inmos Transputer is most related to IXA and most heavily influenced the formation of the architecture[4]. The Transputer family was designed as a building-block component for parallel processing systems, although the processors are more than capable of performing in uniprocessor systems.

Much of the power of the Transputer is found in its four links which are serial data ports in and out of the processor, allowing interconnection to other processors or subsystems. Although the links run at a very high speed (up to 20MHz), their serial nature limits their performance. To some extent this situation is a result of pin limitations (the processor is a 32-bit system). IXA overcomes this by eliminating the pins required for the memory/data interface and reducing the I/O port handshaking pins to a minimum. Unlike IXA, the Transputer also has a standard memory system with address and data buses, and executes a standard fetch cycle (although it is possible to pass instructions over the links).
The Transputer is also a stack-based machine, however, its evaluation stack is much smaller; it is fixed at three elements. It is used primarily as registers for the evaluation of operations and not as a storage space for data as in IXA. To provide high-speed data and program storage, an on-chip memory is used. This is accessed through a workspace pointer which allows it to act as a stack.

Instructions for the Transputer are not of a fixed length. Instead, encodings are provided so that the most commonly used instructions can be stored in an 8-bit field and executed immediately. The processor contains a shift register which allows larger instructions and data to be constructed from multiple smaller words in a sequential manner. This may seem to cause a performance problem, but most instructions executed can be encoded in a single byte, providing fast execution and requiring little memory for program storage.

4.3.0 Applications

No architecture can be created in a vacuum. If there is no need for it, it will not survive beyond being a research endeavor. The development of IXA was driven by a set of design parameters which can be seen emerging in several areas of the computer industry. This section discusses some of these areas and how IXA might be applied to them.
4.3.1 Uniprocessor Applications

IXA's simplicity and ease of integration make it ideal for applications requiring a single microprocessor to carry out standard computing tasks without being bound to the standard system model.

One such application area is in embedded control. This area has long been a strong market for stack-based designs because of their simplicity and small size, traits which IXA exhibits. The architecture's asynchronous and interrupt handling capabilities also make it well-suited for systems which must deal with irregular programmed operations.

IXA will probably never fare well in the desktop workstation market since it is not ideal for running multitasking, multiuser operating systems. It could, however, find a niche as a form of coprocessor. Many recent machines use smaller, simpler processors to handle I/O tasks (which IXA does very well) so that the main processor's resources can be expended on running the operating system.

Similarly, the architecture would probably not fit well in the notebook market. However, it does fit the needs of smaller, portable computing devices like the recently introduced Personal Digital Assistants (PDA's). These small devices perform tasks such as handwriting recognition and data communications, but do not need to support a full operating system. Currently, RISC processors are used
in these designs, although much of the work is really done by specialized Application-Specific Integrated Circuits (ASIC's). An excellent example is the Apple Newton, which uses an Advanced RISC Machines (ARM) 610 RISC processor. Most of its control functions, however, are provided by an ASIC which is a more complex and larger than the processor [2, p. 105]. One note of particular interest is that the ASIC handles compression and decompression of data going between the processor and memory, a function which could be built into a memory interface in an IXA system. Another processor used in such machines is the AT&T Hobbit, which, although it is a RISC design, contains an on-chip stack for executing C-language code efficiently [9]. The direction of technology in the field is toward using simple processor cores which then have a variety of additional functions built into the chip. This is exactly what IXA accomplishes.

4.3.2 Multiprocessor Applications

Another area of intense research is in multiprocessor systems. The Inmos Transputer has become one of the premier multiprocessing microprocessors, and IXA's close relationship to that architecture should make it an excellent candidate for similar applications.

IXA is best suited for use in asynchronous, self-timed array applications such as wave front processors [3]. In such an application, an array of IXA processors
could be interconnected and each programmed independently with a particular function to carry out against data passing through the array. No program controller is necessary. Instead, simple data passing units are required to feed data to the array with no programmed control.

Alternately, the architecture could be put to use in machines executing a common instruction stream against multiple data sets, commonly referred to a Single-Instruction, Multiple-Data (SIMD) machines. In this arrangement, only a single program controller is required to synchronize the entire array.

4.4.0 Issues for Further Research

Most of the effort made in this work was toward the development of the architecture. When the amount of time required to create the two implementation models is considered, very little time was left to do any but the most rudimentary testing. The testing which was done was aimed primarily at validating the designs and some of the key components of the architecture. However, little was done to see how well the architecture performs in real-world situations. The discussion above would tend to indicate that it should perform at least comparably to current designs, and, in some cases, out-perform those designs. The main goal of any future work done with this architecture should be to determine how well it would perform, and for which problems it is best suited.
There are several possible branches such research could take. First, the VHDL model developed here could be used as the basis for a more powerful test bench capable of showing how the processor could be used alone or with other processors. Additional program controller models could be developed to solve various problems.

Since the program controller is the center of an IXA system, its design and efficiency is critical to overall system performance. Various designs could be evaluated and a design methodology developed to help in the development of complete systems. This could also be tested using VHDL models.

The extensibility of the architecture is one of its key features. Whether or not this is a useful feature, if it is too much or too little, and how it can be used effectively should be researched.

Another direction future research could take would be to investigate what technology may be best used to implement IXA processors. Issues such as fabrication technology, logic design methodology, and scale require further work. Along the same lines is the question of developing a VLSI cell which could be used as the core to larger systems. The FPGA approach should not be neglected;
experimentation with larger FPGA's, or FPGA's with different attributes could
provide more information on the possibilities of low-end production of IXA
processors.

Finally, some time needs to be spent considering the programmability of the
processor. How well can current high-level languages be compiled to run on the
architecture? Of particular interest may be the comparison of C, a language which
makes heavy use of stacks, Forth, a stack-based language, and occam, a parallel
processing language, as they apply to IXA.
5 • Conclusions

The architecture presented here is designed to provide basic computing services to a larger, more complex, and often application-specific system. It accomplishes this by taking the approach started with the RISC movement of reducing the complexity of the instruction set (and thereby placing the burden of more complex software development on the compiler) one step further to reducing the complexity of the hardware supporting the core computing functions. Doing so provides the system implementer with a basic interface which can be developed as necessary into a complete system.

The architecture is also extensible and scalable, allowing the system implementer to easily define new features for a specific application while maintaining a level of software compatibility with other designs using the architecture. The ability to implement the architecture at various levels of definition and scale allows for fabrication in a wide variety of technologies, an important factor in cost-sensitive applications.

This work has defined this architecture as an experiment. The implementations performed and discussion provided were demonstrative of the architecture's potential. Further research fabrication, programming and applications for the architecture would help to develop this potential.
References


Appendix A - Architectural Definition

This appendix contains a stand-alone definition for the architecture.
A.1.0 Introduction
The Implementation Extensible Architecture (IXA) is a specification for a computer system in which the basic design philosophy is not only to reduce the instruction set (leaving much of the programming complexity to the compiler), but also to minimize the hardware (leaving the much of the design complexity to the systems implementer). The architecture defines the programming model, instruction set and I/O protocols for the system; all other aspects of the machine are implementation dependent.

A.1.1 IXA Overview
The architecture is designed around an extremely reduced instruction set. This reduction is achieved through the use of a stack machine as the processor. Two hardware stacks are supported: a Data Stack (DS) and an Instruction Stack (IS); only the DS is required. The size of the stacks is not defined and not necessarily the same.

There is no defined memory interface. Instead, the processor has up to four I/O ports which can be externally interfaced to any type of I/O system.

IXA is most different from other architectures in the way it receives instructions for execution. Instead of keeping a program counter and executing a fetch cycle, the processor handles instructions as they arrive at the I/O ports. The task of feeding instructions to the processor and interpreting the results is left to an implementation-defined external controller. A group of instructions can be loaded into the IS for execution, as well. The processor can be set to execute instructions in either mode.

A large number of the available opcodes have been reserved for implementation-defined purposes. These instructions may be used to add functionality to a complete system.

A.2.0 Architecture
IXA takes a multi-layered approach to the architecture. There are three defined layers:
Level 0 defines the absolute minimum requirements of the architecture to provide a useful computing device. In this level of implementation, the IS is not implemented, a minimal DS (at least two elements required) is used, only one port is required, and a subset of the instruction set required to support these functions is implemented. This implementation would be targeted at at systems implemented in low-speed technology where the value of an internal IS is negated by the additional hardware required to support it.

Level 1 defines a system that implements all Level 0 components and adds support for the IS. One or more ports may be implemented. Some instructions which require extensive hardware support (or which would not be of value to the processor due to its limited size) are omitted. This level would be optimized for medium-speed, stand-alone systems.

Level 2 covers the full IXA instruction set and hardware features. A Level 2 implementation should contain at least two ports, and the implementation method and technology should provide high-performance.

At each level, there are three categories of features. Mandated features are the instructions and hardware necessary for an implementation to meet the requirements of a level. For levels 0 and 1, Defined features are those which are defined for higher levels and which can be implemented, but are not required. Finally, Extensible features are those which are added at any level to meet implementation-specific needs. These elements are not defined except in terms of how they relate to the rest of the architecture. An implementer may choose to partition the extensible layer into proprietary layers for system design and integration purposes.

This document defines the standards for all three base levels as well as the Extensible Architecture Interface.

A.2.1 Data Types
Internally, the processor provides support only for a word-sized operand. The size of a word is processor implementation dependent; the minimum is eight bits. Other data types may be externally defined and handled or passed to the processor for processing.

IXA-defined instructions are always eight bits. Data and instructions are differentiated by the state of an additional bit (the Instruction Bit) which is appended in the word_size-plus-one bit position. A set Instruction Bit
indicates an instruction. This bit is not used internally and may be discarded. The I/O ports have the capability of turning this bit on to allow the processor to output instructions.

A.2.2 Operation of the Data Stack
The Data Stack is a circular buffer which as addressed in a LIFO manner by the Data Stack Pointer (DSP). Data may be pushed onto the stack from the I/O ports implicitly or under the control of a program. Data can also be popped from the Instruction Stack on to the DS.

The minimum size of the DS is two elements. The DS can be no larger than the number of elements which can be addressed by a word-sized value in a particular implementation. In general, the DS will either be relatively small (2^32 elements) supporting only one stack frame, or very large (128+ elements) supporting multiple, concurrent stack frames. Other than the ability to directly or relatively set the stack pointer, there is no defined support for multiple stack frames.

In general, operations take one or two items from the DS and may return a result to it, all of which is handled implicitly. The DSP can also be moved relatively or absolutely allowing great flexibility in the use of the stack space.

The stack pointer is initialized to zero. It points to the item which is currently at the TOS. When an item is pushed, the pointer is pre-decremented, and the new item inserted at the new location. For a pop, the pointer is incremented. The items available to immediate operations are those at the current location, and the current location plus one.

A.2.3 Operation of the Instruction Stack
The Instruction Stack functions in a manner similar to the DS, except that it is the implicit destination for instructions arriving at any of the ports during a LOAD operation. ALU operations cannot act upon its contents, nor can some stack manipulation operations.

The main purpose of the IS is to store and execute one or more code segments. Code is loaded into the stack in FIFO order (i.e. in the reverse of the execution order) beginning at the location of the current stack pointer. The load is terminated with the RESUME instruction. The code can be executed at any time with the EXEC instruction. When code is being executed
from the stack, the processor is in stack-based execution mode. Stack-based execution ends with the HALT instruction. With the ADJI and SSPI instructions, jumps, subroutine calls, branches, and returns can all be simulated.

The instruction stack pointer is initially zero. Operation is as above such that (a) when code is pushed on to the stack, the pointer pre-decrements and the code is inserted at the new location, and (b) when the code executed, the pointer is incremented through the program. When executing an instruction, the pointer is incremented either one or two positions (depending on the instruction) immediately after the instruction is read for processing, but before the instruction is executed. This means that any changes the instruction makes to the pointer will occur after it has been incremented.

The IS is not implemented in the Level 0 architecture. This means that all instructions must come through the I/O port(s), and that an external controller must handle any conditional changes in execution by observing the status register.

A.2.4 I/O Ports Operation
IEA allows up to four parallel I/O ports to be incorporated into the processor; only one is required. The I/O ports are controlled by the processor's router which is responsible for differentiating incoming data from instructions and prioritizing the resulting groups for the processor's internal control systems.
Words can arrive at the ports asynchronously. Each port has word-size plus one bits and may be implemented with uni- or bidirectional I/O. If the implementation is uni-directional, the two physical ports (one for input, one for output) are mapped to the same logical port. Ports must be numbered sequentially beginning at zero. Each physical I/O port also has input and output handshaking signals.

To use an I/O bus, the processor or external system asserts its output handshake and places its data on the bus. It must then hold the data and the handshake until receiving an acknowledgment on its input handshake line, at which time it should isolate itself from the bus and drop its handshake. Both handshake lines must become unasserted for a period of time before the bus can be used again. A receiving processor should only use the initial handshake as a data latch signal with the understanding that the data may not be stable at the point it is asserted. Note that the
protocol specifies no handling of collisions on shared busses in keeping with the implementation-oriented nature of the architecture. Instead, the expectation is that if collisions are a possibility, they will be avoided through software control or handled through hardware interfacing.

The I/O Control Register (IOCR) affects how the processor handles incoming instructions and data, and the state of the instruction bits for each port during output. If the Port Lockout Bit (bits 1, 3, 5, 7 of the IOCR) is set for a particular port, then instructions and data arriving at that port are ignored. During normal mode, this simply disables a port. During stack-based execution, if the bit is not set and an instruction arrives, an implicit HALT is executed, followed immediately by the pending instruction. Data arriving under these conditions is available to the next instruction to execute as if it were on the top of the DS. EXEC can be used to restart stack-based execution. This mechanism can be used to perform fast interrupts, although care should be taken not to modify the DS or IS if it is desirable to resume stack-based execution.

The Instruction Bit (bits 0, 2, 4, 6 of the IOCR) determines whether the upper-most bit in the specified I/O port is set during output. This bit can be set to send instructions to other processors or used with other systems as an additional signal line.

Data can be read and written under programmed control with the READ and WRIT operations. The result of a WRIT to a port which has incoming data pending is undefined. Small implementations may choose to treat it as an error condition, while larger implementations may use buffering or queuing of data to deal with the event. If a READ is executed on a port, and no data is available, the processor stalls until the data becomes available. This implements a handshaking or rendezvous protocol for parallel interprocessor communications.

A.2.5 Interrupts
In addition to the ability to interrupt with an instruction as discussed above, standard interrupt operations are supported. There are four interrupt levels which may be signalled; the highest, interrupt level 0, is non-maskable. The other levels may be masked by appropriately setting the interrupt mask bits in the Processor Status Register (PSR, bits 4 and 5). A given interrupt is not masked if it is greater than or equal to the mask. A mask of 00 disables all but interrupt 0, while 11 allows all interrupts. Only the non-maskable, level 0 interrupt is mandated.
When an interrupt occurs, the processor stops executing a program from the IS or accepting any pending inputs. The interrupt is acknowledged by asserting the appropriate IACK signal for the level of the interrupt accepted. Upon IACK being signalled, all devices should withdraw all pending I/O requests (including the requesting device, if appropriate). Additionally, the requesting device should drop its request signal when IACK is asserted. When all I/O is cleared and the IRQ to be serviced is dropped, the processor will push the current status word onto the DS, go to Normal mode, and begin accepting input as normal; IACK will remain asserted. When the requesting device has completed the interrupt function, it can clear the interrupt mode with the CLRINT instruction. This simply causes the IACK signal to be dropped, so that if it is necessary for the pre-interrupt status to be restored, the requesting device must perform a SSW instruction.

Higher-level interrupts are allowed while the processor is already interrupted. Each new interrupt performs the same procedure as above, pushing a new copy of the Status Word on the DS. To disable further interrupts, the first interrupting processor should set the interrupt level to 00 to lockout all but the NMI. Note that interrupts are non-nestable and non-reentrant.

A.2.6 Conditional Execution
Several instructions allow for conditional execution. The given condition (a three-bit code) is matched against the current ALU state flags in the Status Register. If the condition is true, the instruction is executed. Otherwise, the equivalent of a no-op is performed.

The condition codes are:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Condition(mnemonic)</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>On Any (ANY)</td>
<td>any</td>
</tr>
<tr>
<td>001</td>
<td>On Zero (EQ)</td>
<td>Z</td>
</tr>
<tr>
<td>010</td>
<td>On Not Zero (NE)</td>
<td>Z</td>
</tr>
<tr>
<td>011</td>
<td>On Negative (NEG)</td>
<td>N</td>
</tr>
<tr>
<td>100</td>
<td>On Carry (CY)</td>
<td>C</td>
</tr>
<tr>
<td>101</td>
<td>On Overflow (OV)</td>
<td>V</td>
</tr>
<tr>
<td>110</td>
<td>On Greater Than or Equal (GE)</td>
<td>N·V + N·V</td>
</tr>
<tr>
<td>110</td>
<td>On Less Than or Equal (LE)</td>
<td>Z + N·V + N·V</td>
</tr>
</tbody>
</table>
A.2.7 Execution Cycle

The system is initially in the reset state in which the stacks are empty, the stack pointers are set to their reset state, the I/O ports are cleared, the IOCR is zeroed, and the status register is zeroed. From this point, the system begins accepting input on its ports. Words arriving at the ports are prioritized for processing as follows:

1. Instructions arriving at lower-numbered ports have higher priority over those arriving at higher-numbered ports, as well as over all data.

2. Data arriving at lower-numbered ports have priority over data arriving at higher-numbered ports.

One instruction (if available) and any number of data words (if available) are routed to the processor each processor cycle. Since both are simultaneously available, it is possible to load instructions in parallel with loading data, or execute instructions on immediately arriving data. It is the implementer's responsibility to handle concurrency issues.

If the system is in the Normal state, instructions are processed as they become available to the processor from the router. If no instruction is available, any incoming data is pushed onto the DS. If an instruction is executing and data is available to the processor from the router, it appears to the current instruction as if it were pushed onto the stack already (with proper precedence being observed). The number of data items which can be concurrently pushed is defined by the implementation, and may be critical where concurrency is an issue. When an instruction executes in normal mode, the system state temporarily moves to the Executing state; this can be used as an indication of processor activity. Again, an implementation of the processor may allow superscalar execution of instructions which do not have stack dependencies which would prevent proper sequential execution.

When the LOAD instruction is executed, the system moves to the Loading state. In this mode, all incoming instructions are placed on the IS. This continues until the RESUME instruction is executed, which returns the processor to the Normal mode.

If the EXEC instruction is executed, the processor moves to the Executing mode. In this mode, instructions are 'popped' off the IS and executed as if they were arriving from the ports. Execution continues until the HALT instruction is executed, when the system returns to Normal mode.
If the system is in Normal mode and there are no pending instructions or data, it simply waits until there are. This is different from other architectures which require the processor to always be doing something, even if it is a no-op.

The only other state that the system can enter is an error state. If this condition occurs, the processor completely stops and indicates the error in the Status Register as 11; external lines may be used to provide error specification. The only error condition mandated by the architecture is the illegal instruction error. In general, errors found in other processors do not occur in this architecture (i.e. memory-related errors).

The number of cycles required by each operation is not specified by the architecture. The architecture has been designed to optimize certain operations, such as the concurrent loading of data and instructions. Processor implementations should strive to minimize time for all operations.

### A.3.0 Programmer's Model (Level 2)

#### Data Stack

| D0, DTOS | D1 |

#### Instruction Stack

| I0, ITOS | I1 |

#### DS Pointer

| DSP |

#### IS Pointer

| ISP |

#### Port I/O Regs.

| I0 | P0 |
| I1 | P1 |
| I2 | P2 |
| I3 | P3 |

#### IOCR

| L3 | I3 | L2 | I2 | L1 | I1 | L0 | I0 |

#### Status Reg.

| S1 | S0 | M1 | M0 | N | C | V | Z |

#### Register Descriptions:

Data Stack & Instruction Stack: The two hardware stacks. The stacks are circular queues. Overflow is not monitored since, using the Adjust instructions, it is possible to keep several areas of the stack active at once.
Stack Pointers: point to TOS. These can be read to the stack with the PSP and PSPI instructions, and modified relatively with ADJ or absolutely with SSP. The stack is implemented as a circular memory so that the pointers roll over and under as necessary. The pointers may be physically only as large as necessary, but are always viewed by the programmer as word-sized registers.

Port I/O Registers: Word-plus-one bit registers which hold data as it is moved in and out of the system. The router handles these directly, although the programmer can specifically read from and write to any one. The Instruction Bits are stored in the IOCR (for output). The ports may or may not be physically present in the form of data latches.

I/O Control Register (IOCR): Stores I/O mode information. The I bits represent the instruction bits for each of the output ports. The L bits are the instruction lock out control bits. The IOCR may be read with the PIO instruction and set with SIO.

Status Register: Lower 4-bits contain the standard ALU status, bits 4-5 hold the current interrupt mask, and bits 6-7 contain the system status. The status codes are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Normal</td>
</tr>
<tr>
<td>01</td>
<td>Executing</td>
</tr>
<tr>
<td>10</td>
<td>Loading</td>
</tr>
<tr>
<td>11</td>
<td>Error</td>
</tr>
</tbody>
</table>

The interrupt state is indicated by a separate interrupt acknowledgment signal, as is the level of an accepted interrupt. If a particular error indication is required, it is also implemented as separate signals.

A.4.0 Instruction Set
The instructions have been broken into five groups; the first two or three bits of the instruction indicate the group to which it belongs. The groups are:

- Extensible Architecture Group (00)
- Arithmetic Group (010)
- Control Group (011)
- Stack & I/O Group (10)
Stack Adjust Group (11)

Many instructions have several modes of operation. To differentiate these modes, one or more letters is affixed to the basic mnemonic. These affixes are:

- I Apply to the instruction stack
L- Long or large; operand comes from the stack.
-C Constant; maintain the second operand to use as a constant

The I postfix indicates that the instruction is to act on the I Stack. If there is no I postfix, and the instruction does not implicitly operate on the I Stack (such as EXEC), then the instruction operates on the D Stack.

Using the L prefix indicates that the operand for the instruction is to come from the stack. If this operand is to come from the DS, then it is popped off the top of stack. If it comes from the IS, then it is interpreted to be the next word following the instruction. Note that this prefix is not used with operations which always take an argument from the stack, but rather with those which can either take an argument from the stack or use information encoded in the instruction word.

The C postfix is used to indicate the constant mode. In this mode (used with ALU operations), the second argument to the operation is not popped from the stack. This allows it to be used repeatedly as a constant to an operation.

In general, arguments come from the stack(s) or from within the instruction byte. Whenever possible, data has been encoded in the instruction to reduce stack access and usage.

Bits in the instruction are numbered from zero starting at the right (the LSB).

The instruction set presented here contains all of the instructions defined for the architecture. The minimum architecture level associated with each instruction is noted in parentheses after the instruction’s name. Additional instructions may be added either within the processor or in an external processor by using the Extensible Group. Note: the few unused bit patterns remaining are reserved and may not be used.
Appendix A contains the generic instruction formats. Appendix B contains several assembler pseudo-instructions which perform standard stack operations and can be implemented using single IXA instructions.

A.4.1 Extensible Architecture Group (00)
All 64 opcodes prefixed by 00 are reserved for use by external controllers. The only instruction specified by the architecture for this group is EXTOP:

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Bit Pattern</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extended</td>
<td>EXTOP</td>
<td>00000000 w</td>
<td>\textit{w} is word representing \textit{op}</td>
</tr>
</tbody>
</table>

EXTOP causes the next word to be interpreted as the instruction. This allows for \(2^{\text{word size}}\) additional instructions.

A processor which does not use any extended codes should treat all 00 group instructions as NOP instructions.

An implementation which does not support any extended instructions can either treat 00 group instructions as NOPs or as illegal instructions.

A.4.2 Arithmetic Group (010)
The arithmetic group implements the standard arithmetic and logical functions. These operations can only be performed on the DS. In many of these instructions, bit 0 represents constant mode; instructions postfixed with \('C'\) have this bit set and only drop the TOS (the second element is unmodified to be used again).

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Bit Pattern</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not (0)</td>
<td>NOT</td>
<td>01000001</td>
<td></td>
</tr>
</tbody>
</table>

Bit-invert the TOS.

| And (0) | AND      | 01000010    |                   |
|         | ANDC     | 01000011    | \textit{Save DS2} |

Logical AND of DS1 and DS2.
Or (0)          OR
             ORC
01000100
01000101       ;Save DS2

Logical OR of DS1 and DS2.

Xor (0)         XOR
             XORC
01000110
01000111       ;Save DS2

Logical XOR of DS1 and DS2.

Shifts (0)     SLL
               SRL
               SLA
               SRA
               SLR
               SRR
01001000       ;Shift-Left-Logical
01001001       ;Shift-Right-Logical
01001010       ;Shift-Left-Arith.
01001011       ;Shift-Right-Arith.
01001100       ;Shift-Left-Rotate
01001101       ;Shift-Right-Rotate

Shifts DS1 as indicated. Logical shifts cause zeros to be placed in the emptied-bits, and one’s shifted out to set the C flag. Arithmetic shifting causes the sign bit (7) to be preserved. Rotational shifting causes the bit shifted out to fill the emptied bit position. Bit 0 is used as the direction indicator, bits 1 and 2 indicate that the fill-bit source is zeros (00), the sign bit (01), or the carry-out (10).

Add (0)        ADD
               ADDC
01010000
01010001       ;Save DS2

Adds DS1 to DS2.

Subtract (1)   SUB
               SUBC
01010010
01010011       ;Save DS2

Subtracts DS1 from DS2.
Multiply (2)  
MUL 01010100  
MULC 01010101 ;Save DS2 (to DS3)

Multiplies DS1 by DS2. Only the LSW is returned. The multiplication is a signed operation.

Divide (2)  
DIV 01010110  
DIVC 01010111 ;Save DS2

Divides DS2 by DS1, returning the quotient. Overflow occurs if the resulting quotient would exceed the representable range. An error occurs if the dividend is zero. The division is a signed, integer operation.

Compare (2)  
CMP 01011000 ;Fully destructive  
CMPI 01011001 ;Constant mode  
CMPN 01011010 ;Non-destructive

Compare subtracts DS1 from DS2 and sets the condition flags. CMP pops both operands, CMPI pops only the TOS, and CMPN leaves the operands untouched. No result is pushed. In Level 0 and 1 implementations, this function can be performed using combinations of SUB (or ADD and NOT), ADJ and GET.

A.4.3 Control Group (011)  
The control group contains miscellaneous instructions used to test and modify various aspects of processor operation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Bit Pattern</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push Status Word (0)</td>
<td>PSW</td>
<td>01100000</td>
<td></td>
</tr>
</tbody>
</table>

Pushes a copy of the current status word onto the DS.

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Status Word (0)</td>
<td>SSW</td>
<td>01100001</td>
</tr>
</tbody>
</table>

Sets the status word from the DS. Use to modify ALU settings, set interrupt mask, or change processor status.
Halt (1) 

HALT cond 01110ccc ;ccc is the condition

Conditionally stop IS based execution.

Push Stack Pointer (0) PSP (1) PSPI 01100010 01100011 ;DS pointer ;IS pointer

Pushes value of selected stack pointer onto DS. Value of DS for PSP is prior to pushing pointer; value of IS for PSPI is after popping current instruction. PSPI is not used for a Level 0 implementation.

Push IO Mode (0) PIO 01100100

Pushes current IO Mode byte onto DS.

Set IO Mode (0) SIO 01100101

Sets the IO mode from the DS.

Load Instructions (1) LOAD 01100110

Loads instructions from all enabled ports into the IS. An implied PUSH is performed for each incoming instruction. Load is terminated with the RESUME instruction.

Resume Normal (1) RESUME 01100111

End a load and resume immediate execution of instructions as they arrive.

Execute from IS (1) EXEC 01111ccc ;ccc is condition

Conditionally begin executing instructions from IS.
No Operation (0)       NOP       01101000

Perform no operation this cycle. Processor state is unchanged.

Toggle Trace (1)       TTRA      01101001

Toggles the current trace mode state. In stack-based execution, setting trace mode causes the processor to stop after each instruction. An external controller/debugger must then issue an EXEC instruction to perform the next instruction.

Clear Interrupt (1)    CLRINT    01101010

Clears all interrupt acknowledgement lines (IACK), and returns the processor to Normal mode. The pre-interrupt status word is not automatically reset, but must be reloaded by the interrupting process.

A.4.4 Stack Manipulation & I/O Group (10)
The Stack Manipulation group contains instructions to read and write from the four ports under the control of an instruction (normally port input is automatic), as well as move data on the stacks and modify the stack pointers.

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Bit Pattern</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read from Port</td>
<td>READ p</td>
<td>100000pp</td>
<td>.pp is the port</td>
</tr>
</tbody>
</table>

Reads a word from the given port to the DS. An implied PUSH is performed. The processor will wait indefinitely (or until RESET) for the word.

<table>
<thead>
<tr>
<th>Write to Port (0)</th>
<th>WRIT p</th>
<th>100001pp</th>
<th>.pp is the port</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>WRITC p</td>
<td>100010pp</td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>WRITI p,w</td>
<td>100011pp</td>
<td></td>
</tr>
</tbody>
</table>

Writes a word from the DS (WRIT and WRITC) or IS (WRITI) to the given port. An implied POP is performed. WRITI is not implemented in Level 0. The processor will wait indefinitely (or until RESET) for the port’s output queue to accept the word.
Pushes the next word from the IS onto the DS or from the DS on to the IS.

Retrieve Word (0) GET n 1010nnnn ; nnnn is offset
(0)* LGET n, src 10110000 ; n is offset

Copies the word at the given offset to the TOS of the DS. For LGET, the offset comes from the DS and is taken to be the offset from the TOS after the word representing the offset is popped. The assembler must either assemble a PUSH prior to LGET, or provide another means of getting the offset on to the DS.

*LGET is not required in any implementation if the depth of the DS is less than 16 elements.

Swap Words SWAP 10010010

Swaps the top two elements of the DS.

A.4.5 Stack Pointer Adjust Group (11)
There are two instructions (and their variations) in this group. These instructions directly modify the stack pointers, either relatively (ADJ) or absolutely (SSP).

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Bit Pattern</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjust Stack (0)</td>
<td>ADJ cond, off</td>
<td>1100ccc</td>
<td>; ccc is condition,</td>
</tr>
<tr>
<td>Pointer</td>
<td></td>
<td></td>
<td>; n+1 is offset</td>
</tr>
<tr>
<td>(0)*</td>
<td>LADJ cond, off</td>
<td>11100ccc n</td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>ADJI cond, off</td>
<td>1101nccc</td>
<td></td>
</tr>
<tr>
<td>(1)*</td>
<td>LADJI cond, off</td>
<td>11110ccc n</td>
<td></td>
</tr>
</tbody>
</table>

Conditionally adjust the stack pointer by the given offset for the indicated stack. For the long-mode instructions, the offset comes from the DS, and the adjustment is relative to the TOS after the offset is popped (or, in the case of ADJI/LADJI, after the instruction is popped). The assembler must either assemble a PUSH prior to LADJ, or provide another means of getting the offset on to the DS.
*LADJ and LDJ are only required if the IS is present (LADJI) or it is necessary to modify the stack pointer by more than two elements and it is more efficient to do so with a long instruction as opposed to multiple short instructions.

```
Set Stack Pointer (0)    SSP cond, val     11101ccc ;ccc is condition
(1)    SSPI cond, val     11111ccc
```

Conditionally sets selected stack pointer from the DS. The assembler must either assemble a PUSH prior to SSP, or provide another means of providing the offset.
Appendix B - Condition Code Results

This appendix contains a complete listing of the frequencies of various 68000 condition codes used in section 2.5.7. These were determined by lexically scanning single code resources in ten different Macintosh applications. The applications scanned were:

<table>
<thead>
<tr>
<th>Application</th>
<th>Purpose</th>
<th>Resource Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADEExpand</td>
<td>Decompression Utility</td>
<td>6</td>
</tr>
<tr>
<td>TrueType Converter</td>
<td>Font Translation Utility</td>
<td>2</td>
</tr>
<tr>
<td>Spectre</td>
<td>Game</td>
<td>7</td>
</tr>
<tr>
<td>Sound Tracker</td>
<td>Sound Generation Utility</td>
<td>3</td>
</tr>
<tr>
<td>Shufflepuck Cafe</td>
<td>Game</td>
<td>5</td>
</tr>
<tr>
<td>Expert Color Paint</td>
<td>Graphics Application</td>
<td>7</td>
</tr>
<tr>
<td>SoftPC</td>
<td>DOS Emulation</td>
<td>11</td>
</tr>
<tr>
<td>StyleWriter</td>
<td>Printer Driver</td>
<td>-8192</td>
</tr>
<tr>
<td>Think C</td>
<td>C Compiler</td>
<td>7</td>
</tr>
<tr>
<td>ZTerm</td>
<td>Communications</td>
<td>8</td>
</tr>
</tbody>
</table>

A simple text scan was made of each selected resource. Selection of resources was based primarily on size; extremely large resources were unscannable and small ones were of lesser value. No attempt was made to decode the resources.
<table>
<thead>
<tr>
<th>Software</th>
<th>8RA</th>
<th>8NE</th>
<th>8EQ</th>
<th>8HI</th>
<th>BLS</th>
<th>BCC</th>
<th>BCS</th>
<th>8VC</th>
<th>8VS</th>
<th>8PL</th>
<th>8MI</th>
<th>8GE</th>
<th>8LT</th>
<th>8GT</th>
<th>8LE</th>
<th>Totals</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADEexpand</td>
<td>35</td>
<td>20</td>
<td>31</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>95</td>
</tr>
<tr>
<td>TTConverter</td>
<td>6</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>Spectre</td>
<td>17</td>
<td>11</td>
<td>23</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>12</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>SoundTracker</td>
<td>27</td>
<td>18</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>73</td>
</tr>
<tr>
<td>Shufflepuck Cafe</td>
<td>28</td>
<td>34</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expert Color Paint</td>
<td>11</td>
<td>9</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>43</td>
</tr>
<tr>
<td>SoftPC</td>
<td>37</td>
<td>27</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>93</td>
</tr>
<tr>
<td>StyleWriter</td>
<td>39</td>
<td>26</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>93</td>
</tr>
<tr>
<td>Think C</td>
<td>16</td>
<td>31</td>
<td>39</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>ZTerm</td>
<td>11</td>
<td>14</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>257</td>
<td>178</td>
<td>205</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>0</td>
<td>5</td>
<td>16</td>
<td>10</td>
<td>37</td>
<td>28</td>
<td>23</td>
<td>37</td>
<td>820</td>
</tr>
<tr>
<td>Percentage</td>
<td>31.34</td>
<td>21.71</td>
<td>25.00</td>
<td>0.61</td>
<td>0.37</td>
<td>0.73</td>
<td>1.22</td>
<td>0.00</td>
<td>0.61</td>
<td>1.95</td>
<td>1.22</td>
<td>4.51</td>
<td>3.41</td>
<td>2.80</td>
<td>4.51</td>
<td>100</td>
</tr>
<tr>
<td>Rank</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>12</td>
<td>14</td>
<td>11</td>
<td>9</td>
<td>15</td>
<td>13</td>
<td>8</td>
<td>10</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Appendix C - VHDL Source Code

This appendix contains the source code to the three VHDL programs described in section 3.1. The first program represents a two-port, Level 2 implementation of the architecture. The second is a simple program controller for testing the model. The last program is a test bench connecting the first two programs into a single unit.
-- IXAip VHDL Behavioral Model
-- Matthew Dinmore
-- August, 1993

-- This program models a level 2, two port Implementation Extensible
-- Architecture instruction processor. As a behavioral model, no attempt
-- has been made to represent the internal function of any possible
-- implementation. Timing representations are provided only to make
-- physical delays in activity visible.

ENTITY IXAip IS
PORT ( P0i: IN INTEGER; -- Port 0, input
       P0o: OUT INTEGER; -- Port 0, output
       P1i: IN INTEGER;  -- Port 1, input
       P1o: OUT INTEGER; -- Port 1, output
       H10i: IN BIT;     -- Handshake in, port 0 input
       H10o: IN BIT;     -- Handshake in, port 0 output
       H00i: OUT BIT;    -- Handshake out, port 0 input
       H00o: OUT BIT;    -- Handshake out, port 0 output
       H11i: IN BIT;     -- Handshake in, port 1 input
       H11o: OUT BIT;    -- Handshake in, port 1 output
       H10o: OUT BIT;    -- Handshake out, port 1 input
       H110: OUT BIT;    -- Handshake out, port 1 output
       IRQ0: IN BIT;     -- Interrupt request 0
       IRQ1: IN BIT;     -- Interrupt request 1
       IACK0: OUT BIT;   -- Interrupt Acknowledge 0
       IACK1: OUT BIT;   -- Interrupt Acknowledge 1
       SW: OUT INTEGER;  -- Status Word output
       IPRESET : IN BIT  -- Reset control line
     );

CONSTANT kMaxIStack : INTEGER := 64; -- Maximum size of I Stack
CONSTANT kMaxDStack : INTEGER := 8;  -- Maximum size of D Stack
CONSTANT kModeNormal : INTEGER := 0;  -- Mode definitions
CONSTANT kModeExecuting : INTEGER := 1;
CONSTANT kModeLoading : INTEGER := 2;
CONSTANT kModeError : INTEGER := 3;

END IXAip;

ARCHITECTURE IPBehavior OF IXAip IS

-- Int2Bit
-- Converts an input integer to an eight-bit bit_vector
FUNCTION Int2Bit (input : INTEGER) RETURN BIT_VECTOR IS
  VARIABLE retVal : BIT_VECTOR (0 TO 7);
  VARIABLE tempInt : INTEGER;
BEGIN
  tempInt := input;
  FOR i IN 0 TO 7 LOOP
    IF (tempInt MOD 2) = 1 THEN
      retVal(i) := '1';
    ELSE
      retVal(i) := '0';
    END IF;
  tempInt := tempInt / 2;
  END LOOP;
  RETURN retVal;
END Int2Bit;

-- Bit2Int
-- Converts an input 8-bit bit_vector to an integer
FUNCTION Bit2Int (input : BIT_VECTOR) RETURN INTEGER IS
  VARIABLE retVal : INTEGER := 0;
BEGIN
  retVal := 0;
  FOR i IN input'range LOOP
    IF input(i) = '1' THEN

retValue := retValue + (2 ** i);
END IF;
END LOOP;
RETURN retValue;
END Bit2Int;

-- TestCondition
-- Given the current status and a condition from an instruction,
-- determines if the requested condition exists
FUNCTION TestCondition (input, status : INTEGER) RETURN BOOLEAN IS
  VARIABLE temp : INTEGER;
BEGIN
  temp := status MOD 16;
  CASE input IS
    WHEN 0 =>
      RETURN TRUE;
    WHEN 1 =>
      IF temp = 1 THEN
        RETURN TRUE;
      ELSE
        RETURN FALSE;
      END IF;
    WHEN 2 =>
      IF temp /= 1 THEN
        RETURN TRUE;
      ELSE
        RETURN FALSE;
      END IF;
    WHEN 3 =>
      IF temp = 8 THEN
        RETURN TRUE;
      ELSE
        RETURN FALSE;
      END IF;
    WHEN 4 =>
IF temp = 4 THEN
    RETURN TRUE;
ELSE
    RETURN FALSE;
END IF;
WHEN 5 =>
    IF temp = 2 THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    END IF;
WHEN 6 =>
    IF (temp = 0) OR (temp = 1) OR (temp = 4) OR (temp = 5) OR (temp = 10) THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    END IF;
WHEN 7 =>
    IF ((temp >= 1) AND (temp <= 3)) OR ((temp >= 6) AND (temp <= 9)) OR ((temp >= 12) AND (temp <= 13)) THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    END IF;
WHEN OTHERS =>
    RETURN FALSE;
END CASE;
END TestCondition;

SIGNAL IOCR : BIT_VECTOR(0 TO 7) := "00000000";
SIGNAL SR : BIT_VECTOR(0 TO 7) := "00000000";
SIGNAL TraceModeBit : BIT := '0';
-- SIGNAL CurrentMode : INTEGER := kModeNormal;

SIGNAL OutData0 : BIT := '0';
SIGNAL OutData1 : BIT := '0';
SIGNAL DataAvail0 : BIT := 0;
SIGNAL DataAvail1 : BIT := '0';
SIGNAL AcceptData0 : BIT := '0';
SIGNAL AcceptData1 : BIT := '0';
SIGNAL OutputBusy0 : BIT := '0';
SIGNAL OutputBusy1 : BIT := '0';

SIGNAL IntLevel : INTEGER := 4;
SIGNAL IntReq : BIT := '0';
SIGNAL ExecInst : BIT := '0';
SIGNAL StackExec : BIT := '0';

BEGIN

-- Handles asynchronous input and output on port 0
Port0 : PROCESS (Hi0i, Hi0o, OutData0, AcceptData0, IPRESET)
BEGIN

-- Handle IP RESET
IF (IPRESET'EVENT) AND (IPRESET = '1') THEN
  Ho0i <= '0' AFTER 1 ns;
  Ho0o <= '0' AFTER 1 ns;
  DataAvail0 <= '0';
  OutputBusy0 <= '0';
END IF;

-- Respond to incoming data
IF (Hi0i'EVENT) AND (Hi0i = '1') AND (IOCR(1) = '0') AND
(DataAvail0 = '0') THEN
  DataAvail0 <= '1';
END IF;

END;
-- Drop Ack Handshake when sender drops input handshake
IF (Hi0i(EVENT) AND (Hi0i = '0') THEN
  Ho0i <= '0';
END IF;

-- Drop Output Handshake when receiver acks
IF (Hi0o(EVENT) AND (Hi0o = '1') THEN
  Ho0o <= '0';
  OutputBusy0 <= '0';
END IF;

-- Init output handshake on OutData, data placed on port by ExecUnit
IF (OutData0(EVENT) AND (OutData0 = '1') THEN
  Ho0o <= '1';
  OutputBusy0 <= 1;
END IF;

-- Ack sender when ExecUnit Accepts Data, drop DataAvail
IF (AcceptData0(EVENT) AND (AcceptData0 = '1') THEN
  Ho0i <= '1' AFTER 1 ns;
  DataAvail0 <= '0';
END IF;

END PROCESS Port0;

-- Handles asynchronous input and output on port 1
Port1 : PROCESS (Hil, Hilo, OutData1, AcceptData1, IPRESET)
BEGIN
  -- Handle IP RESET
  IF (IPRESET(EVENT) AND (IPRESET = '1') THEN
    Holi <= '0' AFTER 1 ns;
    Holo <= '0' AFTER 1 ns;
    DataAvail1 <= '0';
    OutputBusyl <= '0';
  END IF;
END PROCESS Port1;
END IF;

-- Respond to incoming data
IF (Hili'EVENT) AND (Hili = '1') AND (IOCR(3) = '0') AND
   (DataAvail1 = '0') THEN
   DataAvail1 <= '1';
END IF;

-- Drop Ack Handshake when sender drops input handshake
IF (Hili'EVENT) AND (Hili = '0') THEN
   Holi <= '0';
END IF;

-- Drop Output Handshake when receiver acks
IF (Hilo'EVENT) AND (Hilo = '1') THEN
   Holo <= '0';
   OutputBusy1 <= '0';
END IF;

-- Init output handshake on OutData, data placed on port by ExecUnit
IF (OutData1'EVENT) AND (OutData1 = '1') THEN
   Holo <= '1';
   OutputBusy1 <= '1';
END IF;

-- Ack sender when ExecUnit Accepts Data, drop DataAvail
IF (AcceptData1'EVENT) AND (AcceptData1 = '1') THEN
   Holi <= '1' AFTER 1 ns;
   DataAvail1 <= '0';
END IF;
END PROCESS Port1;

-- Repeat internal status register to external Status Word port
StatusOut : PROCESS (SR)
BEGIN
    SW <= Bit2Int(SR);
END PROCESS StatusOut;

-- Handle interrupts
IRQHandler : PROCESS (IRQ0, IRQ1)
BEGIN
    IF ((IRQ1'EVENT) AND (IRQ1 = '1') AND (SR(4 TO 5) /= "00") AND (IntLevel > 1)) OR
        ((IRQ0'EVENT) AND (IRQ0 = '1')) THEN
        IntReq <= '1';
    END IF;

    IF ((IRQ0'EVENT) AND (IRQ0 = '0')) OR ((IRQ1'EVENT) AND (IRQ1 = '0')) THEN
        IntReq <= '0';
    END IF;
END PROCESS IRQHandler;

-- Handles execution of code and processing of interrupts
ExecUnit :PROCESS
    VARIABLE ISP : INTEGER := 0;
    VARIABLE DSP : INTEGER := 0;
    TYPE InstructionStack IS ARRAY (0 TO kMaxIStack) OF INTEGER;
    TYPE DataStack IS ARRAY (0 TO kMaxDStack) OF INTEGER;
    VARIABLE istack : InstructionStack;
    VARIABLE dstack : DataStack;
    VARIABLE bitTemp : BIT_VECTOR (0 TO 7) := "00000000";
    VARIABLE Result : INTEGER := 0;
    VARIABLE ClkTemp : INTEGER := 0;
    VARIABLE CurrentInstruction : INTEGER := -1;
    VARIABLE D0 : INTEGER := 0;
    VARIABLE D1 : INTEGER := 0;
    VARIABLE CurrentMode : INTEGER := kModeNormal;
BEGIN
-- Handle Instruction Processor Reset signal
IF (IPRESET'EVENT) AND (IPRESET = '1') THEN
  DSP := 0;
  ISP := 0;
  IOCR <= "00000000";
  TraceModeBit <= '0';
  SR <= "00000000";
  CurrentMode := kModeNormal;
  AcceptData0 <= '0';
  AcceptData1 <= '0';
  IntLevel <= 0;
  IACK0 <= '0';
  IACK1 <= '0';
END IF;

-- Handle Processing of interrupts
IF (IntReq'EVENT) AND (IntReq = '1') THEN
  IF (IRQ0 = '1') THEN
    IF (IRQ0 /= '0') THEN
      WAIT UNTIL (IRQ0 = '0') OR (IPRESET = '1');
    END IF;
    IF (Hi0i /= '0') OR (Hili /= '0') THEN
      WAIT UNTIL ((Hii0i = '0') AND (Hii1 = '0')) OR (IPRESET'EVENT);
    END IF;
  ELSIF (IRQ1 = '1') AND (SR(4 TO 5) /= "00") THEN
    IntLevel <= 1;
    DSP := (DSP - 1) MOD kMaxDStack;
    dstack(DSP) := Bit2Int(SR);
    SR(6 TO 7) <= "00";
  END IF;
END IF;
CurrentMode := kModeNormal;
IACK1 <= '1';
IF (IRQ1 /= '0') THEN
   WAIT UNTIL (IRQ1 = '0') OR (IPRESET = '1');
END IF;
IF (Hilo /= '0') OR (Hili /= '0') THEN
   WAIT UNTIL ((Hilo = '0') AND (Hili = '0')) OR (IPRESET'EVENT);
END IF;
END IF;

-- Data load handlers
IF (DataAvail0'EVENT) AND (DataAvail0 = '1') THEN
   IF (P0i > 255) THEN
      IF (CurrentMode = kModeLoading) THEN
         IF (P0i - 256) = 103 THEN
            CurrentMode := kModeNormal;
            SR(6 TO 7) <= "00";
         ELSE
            ISP := (ISP - 1) MOD kMaxIStack;
            istack(ISP) := P0i - 256;
         END IF;
         AcceptData0 <= '1', '0' AFTER 1 ns;
      ELSE
         ExecInst <= '1' AFTER 5 ns, '0' AFTER 6 ns;
      END IF;
   ELSE
      CurrentInstruction := P0i - 256;
      AcceptData0 <= '1', '0' AFTER 1 ns;
   END IF;
ELSE
   DSP := (DSP - 1) MOD kMaxDStack;
   dstack(DSP) := P0i;
   AcceptData0 <= '1', '0' AFTER 1 ns;
END IF;
END IF;
IF (DataAvail1'EVENT) AND (DataAvail1 = '1') THEN
    IF (Pli > 255) THEN
        IF (CurrentMode = kModeLoading) THEN
            IF (Pli - 256) = 103 THEN
                CurrentMode := kModeNormal;
                SR(6 TO 7) <= "00";
            ELSE
                ISP := (ISP - 1) MOD kMaxIStack;
                istack(ISP) := Pli - 256;
            END IF;
            AcceptData1 <= '1 ', '0' AFTER 1 ns;
        ELSEIF (CurrentInstruction = -1) THEN
            CurrentInstruction := Pli - 256;
            AcceptData1 <= '1 ', '0' AFTER 1 ns;
            ExecInst <= '1 AFTER 5 ns, '0' AFTER 6 ns;
        END IF;
    ELSE
        DSP := (DSP - 1) MOD kMaxDStack;
        dstack(DSP) := Pli;
        AcceptData0 <= '1 ', '0' AFTER 1 ns;
    END IF;
END IF;

-- Handle instruction execution events
IF (ExecInst'EVENT AND (ExecInst = '1')) OR ((StackExec'EVENT AND (StackExec = '1'))) THEN
    IF (CurrentMode = kModeExecuting) AND (CurrentInstruction = -1) THEN
        CurrentInstruction := istack(ISP);
        ISP := (ISP + 1) MOD kMaxIStack;
    END IF;

D0 := dstack(DSP);
D1 := dstack((DSP + 1) MOD kMaxDStack);

CASE CurrentInstruction IS
    WHEN 64 => -- NOT
Result := Bit2Int(NOT Int2Bit(D0));
dstack(DSP) := Result;
SR(1 TO 2) <= "00";
WHEN 66|67 => -- AND|ANDC
    Result := Bit2Int(Int2Bit(D0) AND Int2Bit(D1));
    IF (CurrentInstruction = 66) THEN
        DSP := DSP + 1;
    END IF;
    dstack(DSP) := Result;
    SR(1 TO 2) <= "00";
WHEN 68|69 => -- OR|ORC
    Result := Bit2Int(Int2Bit(D0) OR Int2Bit(D1));
    IF (CurrentInstruction = 66) THEN
        DSP := DSP + 1;
    END IF;
    dstack(DSP) := Result;
    SR(1 TO 2) <= "00";
WHEN 70|71 => -- XOR|XORC
    Result := Bit2Int(Int2Bit(D0) XOR Int2Bit(D1));
    IF (CurrentInstruction = 66) THEN
        DSP := DSP + 1;
    END IF;
    dstack(DSP) := Result;
    SR(1 TO 2) <= "00";
WHEN 72 => -- SLL
    Result := (D0 * 2) MOD 256;
    dstack(DSP) := Result;
    SR(1 TO 2) <= "00";
    IF (D0 > 127) THEN
        SR(2) <= '1';
    END IF;
WHEN 73 => -- SRL
    Result := (D0 / 2);
    dstack(DSP) := Result;
    SR(1 TO 2) <= "00";
IF ((D0 MOD 2) = 1) THEN
    SR(2) <= '1';
END IF;
WHEN 74 => -- SLA
    Result := (D0 * 2) MOD 256;
    dstack(DSP) := Result;
    SR(1 TO 2) <= "00";
    IF (D0 > 127) THEN
        SR(2) <= '1';
    END IF;
    IF ((D0 > 127) AND (Result < 128)) OR
    ((D0 < 128) AND (Result > 127)) THEN
        SR(1) <= '1';
    END IF;
WHEN 75 => -- SRA
    Result := (D0 / 2);
    IF (D0 > 127) THEN
        Result := (Result + 128) MOD 256;
    END IF;
    dstack(DSP) := Result;
    SR(1 TO 2) <= "00";
    IF ((D0 MOD 2) = 1) THEN
        SR(2) <= '1';
    END IF;
WHEN 76 => -- SLR
    Result := (D0 * 2) MOD 256;
    SR(1 TO 2) <= "00";
    IF (D0 > 127) THEN
        Result := (Result + 1) MOD 256;
        SR(2) <= '1';
    END IF;
    dstack(DSP) := Result;
WHEN 77 => -- SRR
    Result := (D0 / 2) MOD 256;
    SR(1 TO 2) <= "00";
IF ((D0 MOD 2) = 1) THEN
    Result := (Result + 128) MOD 256;
    SR(2) <= '1';
END IF;
dstack(DSP) := Result;
WHEN 80|81 => -- ADD|ADDC
    SR(1 TO 2) <= "00";
    IF ((D0 + D1) > 255) THEN
        SR(2) <= '1';
    END IF;
    IF (D0 > 127) THEN
        D0 := D0 - 256;
    END IF;
    IF (D1 > 127) THEN
        D1 := D1 - 256;
    END IF;
    Result := D0 + D1;
    IF (Result > 127) OR (Result < -128) THEN
        SR(1) <= '1';
    END IF;
    Result := (Result + 256) MOD 256;
    IF (CurrentInstruction = 80) THEN
        DSP := (DSP + 1) MOD kMaxDStack;
    END IF;
    dstack(DSP) := Result;
WHEN 82|83|88|89|90 => -- SUB|SUBC|CMP|CMPC|CMPN
    SR(1 TO 2) <= "00";
    D0 := Bit2Int(NOT Int2Bit(D0)) + 1;
    IF ((D1 + D0) > 255) THEN
        SR(2) <= '1';
    END IF;
    IF (D0 > 127) THEN
        D0 := D0 - 256;
    END IF;
    IF (D1 > 127) THEN
D1 := D1 - 256;
END IF;
Result := D1 + D0;
IF (Result > 127) OR (Result < -128) THEN
   SR(1) <= '1';
END IF;
Result := (Result + 256) MOD 256;
IF (CurrentInstruction = 82) OR (CurrentInstruction = 89) THEN
   DSP := (DSP + 1) MOD kMaxDStack;
ELSIF (CurrentInstruction = 88) THEN
   DSP := (DSP + 2) MOD kMaxDStack;
END IF;
IF (CurrentInstruction < 88) THEN
   dstack(DSP) := Result;
END IF;
WHEN 84|85 => -- MUL|MULC
   SR(1 TO 2) <= "00";
   IF (D0 > 127) THEN
      D0 := D0 - 256;
   END IF;
   IF (D1 > 127) THEN
      D1 := D1 - 256;
   END IF;
Result := D0 * D1;
Result := (Result + 256) MOD 256;
IF (CurrentInstruction = 84) THEN
   DSP := (DSP + 1) MOD kMaxDStack;
END IF;
 dstack(DSP) := Result;
WHEN 86|87 => -- DIV|DIVC
   SR(1 TO 2) <= "00";
   IF (D0 > 127) THEN
      D0 := D0 - 256;
   END IF;
   IF (D1 > 127) THEN
D1 := D1 - 256;
END IF;
Result := D1 / D0;
IF (Result > 127) OR (Result < -128) THEN
  SR(1) <= '1';
END IF;
Result := (Result + 256) MOD 256;
IF (CurrentInstruction = 86) THEN
  DSP := (DSP + 1) MOD kMaxDStack;
END IF;
dstack(DSP) := Result;
WHEN 96
  => -- PSW
  DSP := (DSP - 1) MOD kMaxDStack;
dstack(DSP) := Bit2Int(SR);
WHEN 97
  => -- SSW
  DSP := (DSP + 1) MOD kMaxDStack;
  SR <= Int2Bit(D0);
CASE SR(6 TO 7) IS
  WHEN "00" => CurrentMode := kModeNormal;
  WHEN "10" => CurrentMode := kModeExecuting;
  WHEN "01" => CurrentMode := kModeLoading;
  WHEN "11" => CurrentMode := kModeError;
END CASE;
WHEN 112 TO 119
  => -- HALT
  IF (TestCondition((CurrentInstruction MOD 8),Bit2Int(SR)) = TRUE) THEN
    CurrentMode := kModeNormal;
    SR(6 TO 7) <= "00";
  END IF;
WHEN 98
  => -- PSP
  Result := DSP;
  DSP := (DSP - 1) MOD kMaxDStack;
dstack(DSP) := Result;
WHEN 99
  => -- PSPI
  Result := ISP;
  DSP := (DSP - 1) MOD kMaxDStack;
dstack(DSP) := Result;

WHEN 100 => -- PIO
    Result := Bit2Int(IOCR);
    DSP := (DSP - 1) MOD kMaxDStack;
    dstack(DSP) := Result;

WHEN 101 => -- SIO
    IOCR <= Int2Bit(D0);
    DSP := (DSP + 1) MOD kMaxDStack;

WHEN 102 => -- LOAD
    CurrentMode := kModeLoading;
    SR(6 TO 7) <= "01";

WHEN 103 => -- RESUME
    CurrentMode := kModeNormal;
    SR(6 TO 7) <= "00";

WHEN 120 TO 127 => -- EXEC
    IF (TestCondition((CurrentInstruction MOD 8), Bit2Int(SR)) = TRUE) THEN
        CurrentMode := kModeExecuting;
        SR(6 TO 7) <= "10";
    END IF;

WHEN 104 => -- NOP
    -- Nothing, naturally

WHEN 105 => -- TTRA
    TraceModeBit <= NOT TraceModeBit;

WHEN 106 => -- CLINT
    IntLevel <= 4;
    IACK0 <= '0';
    IACK1 <= '0';

WHEN 128 TO 131 => -- READ
    IF (CurrentInstruction MOD 2) = 0 THEN
        IF (DataAvail10 /= '1') THEN
            WAIT UNTIL (DataAvail10 = '1') OR (IPRESET'EVENT);
        END IF;
        IF (IPRESET = '0') THEN
            DSP := (DSP - 1) MOD kMaxDStack;
            dstack(DSP) := P01;
AcceptData0 <= '1', '0' AFTER 1 ns;
END IF;
ELSE
  IF (DataAvail1 /= '1') THEN
    WAIT UNTIL (DataAvail1 = '1') OR (IPRESET'EVENT);
  END IF;
  IF (IPRESET = '0') THEN
    DSP := (DSP - 1) MOD kMaxDStack;
    dstack(DSP) := P1i;
    AcceptData1 <= '1', '0' AFTER 1 ns;
  END IF;
END IF;
WHEN 132 TO 139 => -- WRIT|WRITC
  IF (CurrentInstruction MOD 2) = 0 THEN
    IF (OutputBusy0 /= '0') THEN
      WAIT UNTIL (OutputBusy0 = '0') OR (IPRESET'EVENT);
    END IF;
    IF (IPRESET = '0') THEN
      IF (IOCR(0) = '0') THEN
        P0o <= D0;
      ELSE
        P0o <= D0 + 256;
      END IF;
      OutData0 <= 1, '0' AFTER 1 ns;
    END IF;
  END IF;
ELSE
  IF (OutputBusy1 /= '0') THEN
    WAIT UNTIL (OutputBusy1 = '0') OR (IPRESET'EVENT);
  END IF;
  IF (IPRESET = '0') THEN
    IF (IOCR(2) = '0') THEN
      P1o <= D0;
    ELSE
      P1o <= D0 + 256;
    END IF;
    OutData1 <= 1, '0' AFTER 1 ns;
  END IF;
END IF;
ELSE
  Plo <= D0 + 256;
END IF;
OutData1 <= '1', '0' AFTER 1 ns;
IF (CurrentInstruction < 136) THEN
  DSP := (DSP + 1) MOD kMaxDStack;
END IF;
END IF;
END IF;
WHEN 140 TO 143 => -- WRITI
  IF (CurrentInstruction MOD 2) = 0 THEN
    IF (OutputBusy0 /= '0') THEN
      WAIT UNTIL (OutputBusy0 = '0') OR (IPRESET(EVENT);
    END IF;
    IF (IPRESET = '0') THEN
      P00 <= istack(ISP);
      OutData0 <= '1', '0' AFTER 1 ns;
      ISP := (ISP + 1) MOD kMaxIStack;
    END IF;
  ELSE
    IF (OutputBusy1 /= '0') THEN
      WAIT UNTIL (OutputBusy1 = '0') OR (IPRESET(EVENT);
    END IF;
    IF (IPRESET = '0') THEN
      P10 <= istack(ISP);
      OutData1 <= '1', '0' AFTER 1 ns;
      ISP := (ISP + 1) MOD kMaxIStack;
    END IF;
  END IF;
END IF;
WHEN 144 => -- PUSH
  Result := istack(ISP);
  ISP := (ISP + 1) MOD kMaxIStack;
  DSP := (DSP - 1) MOD kMaxDStack;
  dstack(DSP) := Result;
WHEN 145 => -- PUSHI
Result := dstack(DSP);
ISP := (ISP - 1) MOD kMaxIStack;
DSP := (DSP + 1) MOD kMaxDStack;
istack(ISP) := Result;

WHEN 160 TO 175 => -- GET
Result := dstack((DSP + (CurrentInstruction MOD 16)) MOD kMaxDStack);
DSP := (DSP - 1) MOD kMaxDStack;
dstack(DSP) := Result;

WHEN 176 => -- LGET
Result := dstack((DSP + D0 + 1) MOD kMaxDStack);
dstack(DSP) := Result;

WHEN 146 => -- SWAP
Result := dstack(DSP);
dstack(DSP) := D1;
dstack((DSP + 1) MOD kMaxDStack) := D0;

WHEN 192 TO 207 => -- ADJ
IF (TestCondition((CurrentInstruction MOD 8),Bit2Int(SR)) = TRUE) THEN
    bitTemp := Int2Bit(CurrentInstruction);
    IF (bitTemp(3) = '0') THEN
        DSP := (DSP + 1) MOD kMaxDStack;
    ELSE
        DSP := (DSP + 2) MOD kMaxDStack;
    END IF;
END IF;

WHEN 224 TO 231 => -- LADJ
IF (TestCondition((CurrentInstruction MOD 8),Bit2Int(SR)) = TRUE) THEN
    DSP := (DSP + D0 + 2) MOD kMaxDStack;
END IF;

WHEN 208 TO 223 => -- ADJI
IF (TestCondition((CurrentInstruction MOD 8),Bit2Int(SR)) = TRUE) THEN
    bitTemp := Int2Bit(CurrentInstruction);
    IF (bitTemp(3) = '0') THEN
        ISP := (ISP + 1) MOD kMaxIStack;
    ELSE
        ISP := (ISP + 2) MOD kMaxIStack;
    END IF;
END IF;
WHEN 240 TO 247 => -- LADJ
IF (TestCondition((CurrentInstruction MOD 8), Bit2Int(SR)) = TRUE) THEN
END IF;
WHEN 248 TO 255 => -- XSP
IF (TestCondition((CurrentInstruction MOD 8), Bit2Int(SR)) = TRUE) THEN
END IF;
WHEN OTHERS => -- Illegal Instruction!!
END CASE;
ELSE SR(0) = '0';
ELSE SR(3) = '1';
END IF;

-- If in trace mode and not an EXEC instruction, goto NORMAL Mode
IF (TraceMode = '1') AND (CurrentInstruction > 127) THEN
CurrentMode = 'Normal';
END IF;
SR(6 TO 7) <= "00";
END IF;

-- Clear the instruction register
CurrentInstruction := -1;

-- Initiate another execution cycle if in Executing mode
IF CurrentMode = kModeExecuting THEN
    StackExec <= '0' AFTER 1 ns, '1' AFTER 5 ns, '0' AFTER 11 ns;
END IF;

END IF;
WAIT ON DataAvail0, DataAvail1, IPRESET, IntReq, ExecInst, StackExec;
END PROCESS ExecUnit;

END IPBehavior;
-- IXA Program Controller Model
-- Matthew Dinmore
-- September, 1993

LIBRARY STD;
USE STD.TEXTIO.all;

ENTITY IXApC IS
    PORT ( P0i: IN INTEGER; -- Port 0, input
            P0o: OUT INTEGER; -- Port 0, output
            H0i: IN BIT; -- Handshake in, port 0 input
            H0o: OUT BIT; -- Handshake out, port 0 input
            H1i: IN BIT; -- Handshake in, port 1 input
            H1o: OUT BIT; -- Handshake out, port 1 input
            P1i: IN INTEGER; -- Port 1, input
            P1o: OUT INTEGER; -- Port 1, output
            SW: IN INTEGER; -- Status Word input
            LOAD: IN BIT; -- Program Load Signal
            IPRESET: OUT BIT; -- Processor Reset Line
            PCRESET: IN BIT -- Program Controller Reset
        );
END IXApC;

ARCHITECTURE PCBehavior OF IXApC IS
    SIGNAL Outdata0, Outdata1: BIT := '0';
    SIGNAL Reg0in, Reglin: INTEGER := 0;
    SIGNAL DataAvail0, DataAvail1: BIT := '0';
    SIGNAL Dack0, Dack1: BIT := '0';
    SIGNAL CLK: BIT := '0';
    SIGNAL OutputBusy0, OutputBusy1: BIT := '0';
BEGIN
-- Handles asynchronous input and output on port 0
Port0 : PROCESS (Hi0i, Hi0o, OutData0, PCRESET, Dack0)
BEGIN

-- Handle PCRESET
IF (PCRESET'EVENT) AND (PCRESET = '1') THEN
    Ho0i <= '0' AFTER 5 ns;
    Ho0o <= '0' AFTER 5 ns;
    Reg0in <= 0;
    OutputBusy0 <= '0';
END IF;

-- Respond to incoming data
IF (Hi0i'EVENT) AND (Hi0i = '1') THEN
    Reg0in <= P0i;
    Ho0i <= '1' AFTER 5 ns;
    DataAvail0 <= '1';
END IF;

-- Drop Ack Handshake when sender drops input handshake
IF (Hi0i'EVENT) AND (Hi0i = '0') THEN
    Ho0i <= '0';
END IF;

-- Drop Output Handshake when receiver acks
IF (Hi0o'EVENT) AND (Hi0o = '1') THEN
    Ho0o <= '0';
    OutputBusy0 <= '0';
END IF;

-- Init output handshake on OutData, Data placed in Reg0out by PC
IF (OutData0'EVENT) AND (OutData0 = '1') THEN
    Ho0o <= '1' AFTER 5 ns;
    OutputBusy0 <= '1';
END IF;
IF (Dack0'EVENT) AND (Dack0 = '1') THEN
    DataAvail0 <= '0';
END IF;

END PROCESS Port0;

-- Handles asynchronous input and output on port 1
Port1 : PROCESS (Hilo, Hilo, OutData1, PCRESET, Dack1)
BEGIN
   -- Handle PCRESET
   IF (PCRESET'EVENT) AND (PCRESET = '1') THEN
      Holi <= '0' AFTER 5 ns;
      Holo <= '0' AFTER 5 ns;
      Reglin <= 0;
      OutputBusy1 <= '0';
   END IF;

   -- Respond to incoming data
   IF (Hilo'EVENT) AND (Hilo = '1') THEN
      Reglin <= P0i;
      Holi <= '1' AFTER 5 ns;
      DataAvail1 <= '1';
   END IF;

   -- Drop Ack Handshake when sender drops input handshake
   IF (Hilo'EVENT) AND (Hilo = '0') THEN
      Holi <= '0';
   END IF;

   -- Drop Output Handshake when receiver acks
   IF (Hilo'EVENT) AND (Hilo = '1') THEN
      Holo <= '0';
      OutputBusy1 <= '0';
   END IF;
-- Init output handshake on OutData, Data placed in Reglout by PC
IF (Outdata EVENT) AND (Outdata = '1') THEN
    Holo <= '1' AFTER 5 ns;
    OutputBusy1 <= '1';
END IF;

IF (Dack1 EVENT) AND (Dack1 = '1') THEN
    DataAvail1 <= '0';
END IF;

END PROCESS Port1;

ProgramController : PROCESS
  TYPE MemoryArray IS ARRAY (0 TO 1024) OF INTEGER;
  TYPE PCMode IS (ModeHalted, ModeAttended, ModeUnattended, ModeLoading);
  VARIABLE CurrentMode : PCMode := ModeUnattended;
  VARIABLE LoadFlag : BIT := '0';
  VARIABLE Memory : MemoryArray;
  VARIABLE AR : INTEGER := 0;
  VARIABLE CurrentInstruction : INTEGER := 0;
  VARIABLE BasePort : INTEGER := 0;
  FILE InputFile : TEXT IS IN "TEST.IN"; -- name input file TEST.IN
  VARIABLE InputLine : LINE;
BEGIN
IF (LOAD EVENT) AND (LOAD = '1') THEN
  -- set AR to 0
  AR := 0;
  CurrentMode := ModeUnattended;
  CurrentInstruction := 0;
  -- open input file
  WHILE NOT ENDFILE(InputFile) LOOP
    READLINE(InputFile, InputLine);
    READ(InputLine, CurrentInstruction);
    Memory(AR) := CurrentInstruction;
  END WHILE
END IF;

END ProgramController;
AR := AR+1;
END LOOP;
AR := 0;
IPRESET <= '1', '0' AFTER 5 ns;
WAIT FOR 5 ns;
CLK <= '1', '0' AFTER 5 ns;
END IF;

IF (CLK'EVENT) AND (CLK = '1') AND (CurrentMode = ModeUnattended) THEN
  IF (OutputBusy0 = '0') THEN
    CurrentInstruction := Memory(AR);
    AR := AR + 1;
    IF (CurrentInstruction = 358) THEN
      LoadFlag := '1';
    ELSEIF (CurrentInstruction = 359) THEN
      LoadFlag := '0';
    END IF;
  END IF;

  IF (CurrentInstruction < 320) AND (CurrentInstruction > 255) AND
      (LoadFlag = '0') THEN
    -- Reserve for SMI Instructions
    CASE CurrentInstruction IS
      WHEN 272 => -- LAR
        AR := Memory(AR);
      WHEN 257 => -- WAIT
        CurrentMode := ModeAttended;
      WHEN 276 TO 277 => -- WATIP
        IF CurrentInstruction = 276 THEN
          WAIT UNTIL ((Hi0i(EVENT) AND (Hi0i = '1')) OR
                  (PCRESET'EVENT));
        ELSE
          WAIT UNTIL ((Hi1i(EVENT) AND (Hi1i = '1')) OR
                  (PCRESET'EVENT));
        END IF;
      WHEN 258 => -- WAITV
      WHEN OTHERS =>
    END CASE;
  END IF;
WAIT UNTIL (Reg0in = Memory(AR)) OR (Reglin = Memory(AR))
OR (PCRESET'EVENT);
AR := AR + 1;
WHEN 259 => -- WAITS
WAIT UNTIL (SW = Memory(AR)) OR (PCRESET'EVENT);
AR := AR + 1;
WHEN 280 TO 281 => -- XFERO
IF CurrentInstruction = 280 THEN
   Memory(AR) := Reg0in;
   AR := AR + 1;
   Dack0 <= '1', '0' AFTER 5 ns;
ELSE
   Memory(AR) := Reglin;
   AR := AR + 1;
   Dack1 <= '1', '0' AFTER 5 ns;
END IF;
WHEN 284 TO 285 => -- XFERI
IF CurrentInstruction = 284 THEN
   P0o <= Memory(AR);
   Outdata0 <= '1', '0' AFTER 5 ns;
   AR := AR + 1;
ELSE
   P1o <= Memory(AR);
   Outdata1 <= '1', '0' AFTER 5 ns;
   AR := AR + 1;
END IF;
WHEN 264 => -- AUTO
   CurrentMode := ModeUnattended;
WHEN 265 => -- PRORESET
   IPRESET <= '1', '0' AFTER 5 ns;
WHEN 266 => -- STADDR
   Memory(AR) := AR;
   AR := AR + 1;
WHEN 268 TO 269 => -- XADDR
   IF CurrentInstruction = 268 THEN
P0o <= AR;
Outdata0 <= '1', '0' AFTER 5 ns;
ELSE
P1o <= AR;
Outdata1 <= '1', '0' AFTER 5 ns;
END IF;
WHEN OTHERS =>
-- Bad SMI Instruction
END CASE;
ELSIF (CurrentInstruction = -1) THEN
CurrentMode := ModeHalted;
ELSE
-- Normal data or instruction, pass to port
P0o <= CurrentInstruction;
Outdata0 <= '1', '0' AFTER 5 ns;
END IF;
END IF;
IF (CurrentMode /= ModeHalted) THEN
WAIT FOR 5 ns;
CLK <= '1', '0' AFTER 5 ns;
END IF;
END IF;

IF (CLK'EVENT) AND (CLK = '1') AND (CurrentMode = ModeAttended) THEN
IF (DataAvail0='1') OR (DataAvail1='1') THEN
IF (DataAvail0 = '1') THEN
CurrentInstruction := Reg0in;
ELSE
CurrentInstruction := Reglin;
END IF;

IF (CurrentInstruction < 320) AND (CurrentInstruction > 255) THEN
-- Reserve for SMI Instructions
IF (DataAvail0 = '1') THEN
Dack0 <= '1', '0' AFTER 5 ns;
ENDIF;
ELSE
  Dack1 <= '1', '0' AFTER 5 ns;
END IF;
CASE CurrentInstruction IS
  WHEN 272 => -- LAR
    AR := Memory(AR);
  WHEN 257 => -- WAIT
    CurrentMode := ModeAttended;
  WHEN 276 TO 277 => -- WAIT
    IF CurrentInstruction = 276 THEN
      WAIT UNTIL ((Hi0i'EVENT) AND (Hi0i = '1')) OR (PCRESET'EVENT);
    ELSE
      WAIT UNTIL ((Hi1i'EVENT) AND (Hi1i = '1')) OR (PCRESET'EVENT);
    END IF;
END WHEN;
  WHEN 258 => -- WAITV
    WAIT UNTIL (Reg0in = Memory(AR)) OR (Reglin = Memory(AR)) OR (PCRESET'EVENT);
    AR := AR + 1;
  WHEN 259 => -- WAITS
    WAIT UNTIL (SW = Memory(AR)) OR (PCRESET'EVENT);
    AR := AR + 1;
  WHEN 280 TO 281 => -- XFERO
    IF CurrentInstruction = 280 THEN
      Memory(AR) := Reg0in;
      AR := AR + 1;
      Dack0 <= '1', '0' AFTER 5 ns;
    ELSE
      Memory(AR) := Reglin;
      AR := AR + 1;
      Dack1 <= '1', '0' AFTER 5 ns;
    END IF;
END WHEN;
  WHEN 284 TO 285 => -- XFERI
    IF CurrentInstruction = 284 THEN
P0o <= Memory(AR);
    Outdata0 <= '1', '0' AFTER 5 ns;
    AR := AR + 1;
ELSE
P1o <= Memory(AR);
    Outdata1 <= '1', '0' AFTER 5 ns;
    AR := AR + 1;
END IF;
WHEN 264 => -- AUTO
    CurrentMode := ModeUnattended;
WHEN 265 => -- PROCRESET
    IPRESET <= '1', '0' AFTER 5 ns;
WHEN 266 => -- STADDR
    Memory(AR) := AR;
    AR := AR + 1;
WHEN 268 TO 269 => -- XADDR
    IF CurrentInstruction = 268 THEN
        P0o <= AR;
        Outdata0 <= '1', '0' AFTER 5 ns;
    ELSE
        P1o <= AR;
        Outdata1 <= '1', '0' AFTER 5 ns;
    END IF;
END CASE;
-- Bad SMI Instruction
ELSIF (CurrentInstruction = -1) THEN
    CurrentMode := ModeHalted;
ELSE
    -- Do nothing in attended mode, not in instruction
END IF;
END IF;
IF (CurrentMode /= ModeHalted) THEN
    WAIT FOR 5 ns;
    CLK <= '1', '0' AFTER 5 ns;
END IF;
END IF;

IF (PCRESET'EVENT) AND (PCRESET = '1') THEN
   FOR AR IN 0 TO 1023 LOOP
      Memory(AR) := 0;
   END LOOP;
   AR := 0;
END IF;

WAIT ON LOAD, PCRESET, CLK;
END PROCESS ProgramController;
END PCBehavior;
-- IXAtest
-- Testbench for IXAip and IXApC

USE WORK.IXAip.all;
USE WORK.IXApC.all;

ENTITY IXAtest IS
    port (START, RESETPC : IN BIT);
END IXAtest;

ARCHITECTURE Structure OF IXAtest IS
    SIGNAL Port0pc2ip, Port0ip2pc, SWbus : INTEGER;
    SIGNAL Port1pc2ip, Port1ip2pc : INTEGER;
    SIGNAL RESETip : BIT;
    SIGNAL PC2IP0init, PC2IP0ack, IP2PC0init, IP2PC0ack : BIT;
    SIGNAL PC2IP1init, PC2IP1ack, IP2PC1init, IP2PC1ack : BIT;
    SIGNAL DummyInt0 : INTEGER := 0;
    SIGNAL DummyInt1 : INTEGER := 0;
    SIGNAL DummyBit : BIT := '0';
    SIGNAL DummyBit0 : BIT := '0';
    SIGNAL DummyBit1 : BIT := '0';
    SIGNAL DummyBit2 : BIT := '0';
    SIGNAL DummyBit3 : BIT := '0';

    COMPONENT IXAip
        PORT (P0i : IN INTEGER;
        P0o : OUT INTEGER;
        Pli : IN INTEGER;
        Pl0 : OUT INTEGER;
        Hi0i : IN BIT;
        Ho0i : OUT BIT;
        Hi0o : IN BIT;
        Ho0o : OUT BIT;
        Hi1i : IN BIT;
        Ho1i : OUT BIT;
Hi0 : IN BIT;
H0i : OUT BIT;
IRQ0 : IN BIT;
IRQ1 : IN BIT;
IACK0 : OUT BIT;
IACK1 : OUT BIT;
SW : OUT INTEGER;
IPRESET : IN BIT

END COMPONENT;

COMPONENT IXAp
PORT (P0i : IN INTEGER;
P0o : OUT INTEGER;
P1i : IN INTEGER;
P1o : OUT INTEGER;
Hi0i : IN BIT;
H00i : OUT BIT;
Hi0o : IN BIT;
H00o : OUT BIT;
H1i : IN BIT;
H0i : OUT BIT;
H1o : IN BIT;
H0o : OUT BIT;
SW : IN INTEGER;
LOAD : IN BIT;
IPRESET : OUT BIT;
PCRESET : IN BIT

END COMPONENT;
BEGIN

PC : IXAp PORT MAP (Port0ip2pc,
Port0pc2ip,
Port1ip2pc,
Port2ip2pc,
Port3ip2pc,
Port4ip2pc,
Port5ip2pc,
Port6ip2pc,
Port7ip2pc,
Port8ip2pc,
Port9ip2pc,
Port10ip2pc,
Port11ip2pc,
Port12ip2pc,
Port13ip2pc,
Port14ip2pc,
Port15ip2pc,
Port16ip2pc,
Port17ip2pc,
Port18ip2pc,
Port19ip2pc,
Port20ip2pc,
Port21ip2pc,
Port22ip2pc,
Port23ip2pc,
Port24ip2pc,
Port25ip2pc,
Port26ip2pc,
Port27ip2pc,
Port28ip2pc,
Port29ip2pc,
Port30ip2pc,
Port31ip2pc,
Port32ip2pc,
Port33ip2pc,
Port34ip2pc,
Port35ip2pc,
Port36ip2pc,
Port37ip2pc,
Port38ip2pc,
Port39ip2pc,
Port40ip2pc,
Port41ip2pc,
Port42ip2pc,
Port43ip2pc,
Port44ip2pc,
Port45ip2pc,
Port46ip2pc,
Port47ip2pc,
Port48ip2pc,
Port49ip2pc,
Port50ip2pc,
Port51ip2pc,
Port52ip2pc,
Port53ip2pc,
Port54ip2pc,
Port55ip2pc,
Port56ip2pc,
Port57ip2pc,
Port58ip2pc,
Port59ip2pc,
Port60ip2pc,
Port61ip2pc,
Port62ip2pc,
Port63ip2pc,
Port64ip2pc,
Port65ip2pc,
Port66ip2pc,
Port67ip2pc,
Port68ip2pc,
Port69ip2pc,
Port70ip2pc,
Port71ip2pc,
Port72ip2pc,
Port73ip2pc,
Port74ip2pc,
Port75ip2pc,
Port76ip2pc,
Port77ip2pc,
Port78ip2pc,
Port79ip2pc,
Port80ip2pc,
Port81ip2pc,
Port82ip2pc,
Port83ip2pc,
Port84ip2pc,
Port85ip2pc,
Port86ip2pc,
Port87ip2pc,
Port88ip2pc,
Port89ip2pc,
Port1pc2ip, 
IP2PC0init, 
IP2PC0ack, 
PC2IP0ack, 
PC2IP0init, 
IP2PC1init, 
IP2PC1ack, 
PC2IP1ack, 
PC2IP1init, 
SWbus, 
START, 
RESETip, 
RESETpc 
)

IP : IXAip PORT MAP (Port0pc2ip, 

Port0ip2pc, 
Port1pc2ip, 
Portlip2pc, 
PC2IP0init, 
PC2IP0ack, 
IP2PC0ack, 
IP2PC0init, 
PC2IP1init, 
PC2IP1ack, 
IP2PC1ack, 
IP2PC1init, 
DummyBit, 
DummyBit2, 
DummyBit, 
DummyBit3, 
SWbus, 
RESETip 
)

END Structure;
Appendix D - Level 0 Test Program

This appendix contains the source code and assembled listings of the IXA Level 0 test program which tests all Level 0 instructions and several variations. This was used with both the VHDL and FPGA systems for verification.
; IXA Level 0 Test Program

; All Level 0 instructions are tested.

PROCRESET
; PROCESSOR I/O TEST SET
DATA 01 ; AUTOMATICALLY PUSH 1
WRITC 0 ; SHOULD WRITE 1
DATA 02 ; AUTOMATICALLY PUSH 2
WRITC 0 ; SHOULD WRITE 2
READ 0 ; READ PORT 0
DATA 03 ; DATA TO BE FED TO PORT 0
WRITC 0 ; SHOULD WRITE 3
READ 0 ; READ PORT 0
DATA 04 ; DATA TO BE FED TO PORT 0
WRIT 0 ; SHOULD WRITE 4
WRIT 0 ; SHOULD WRITE 3
WRIT 0 ; SHOULD WRITE 2
WRIT 0 ; SHOULD WRITE 1
WRIT 0 ; SHOULD WRITE 4 - TEST WRAP-AROUND

; PROCESSOR ALU TEST SET
DATA 01
DATA 01
NOT
WRITC 0 ; SHOULD WRITE FE
ADD
WRITC 0 ; SHOULD WRITE FF
DATA 0
ANDC
WRIT 0 ; SHOULD WRITE 0
DATA &HFF
AND
WRITC 0 ; SHOULD WRITE FF
DATA 0
DATA 0
OR
WRITC 0 ; SHOULD WRITE 0
ORC
WRITC 0 ; SHOULD WRITE FF
DATA &HFF
XORC
WRITC 0 ; SHOULD WRITE 0
XOR
WRITC 0 ; SHOULD WRITE FF
DATA &HAA
SLL
WRITC 0 ; SHOULD WRITE 54
SRL
WRITC 0 ; SHOULD WRITE 2A
DATA &HAA
SLA
WRITC ; SHOULD WRITE 54
SRA
WRITC ; SHOULD WRITE 2A
DATA &HAA
SLR
WRITC 0 ;SHOULD WRITE 55
SLR
WRITC 0 ;SHOULD WRITE AA
SRR
WRITC 0 ;SHOULD WRITE 55
SRR
WRITC 0 ;SHOULD WRITE AA

;PROCESSOR CONTROL TEST SET
PROCRESET ;RESET PROCESSOR
PSW
WRITC 0 ;SHOULD WRITE 0
DATA &HFF
DATA &HFF
ADD
PSW
WRIT 0 ;SHOULD WRITE 0C (NEGATIVE, CARRY)
DATA 0
SSW
PSW
WRIT 0 ;SHOULD WRITE 0
PIO
WRIT 0 ;SHOULD WRITE 0
DATA 1
S1O ;INSTRUCTION BIT OF PORT 0 SHOULD BECOME SET
PIO
WRIT 0 ;SHOULD WRITE 1

;PROCESSOR STACK MANIPULATION TEST SET
PSP
WRITC 0 ;WRITES CURRENT STACK POINTER FOR BASELINE TEST
PSP
WRIT 0 ;WRITES NEW STACK POINTER -
;SHOULD BE ONE LESS THAN ABOVE
DATA 01 ;FILL STACK WITH DATA
DATA 02
DATA 03 ;STACK POINTER LEFT POINTING AT 3
WRITC 0 ;SHOULD WRITE 3
GET 0
WRIT 0 ;SHOULD WRITE 3
GET 1
WRIT 0 ;SHOULD WRITE 2
PSP
SSP ANY
WRITC 0 ;SHOULD WRITE 3
SWAP
WRIT 0 ;SHOULD WRITE 2
WRIT 0 ;SHOULD WRITE 3
WRIT 0 ;SHOULD WRITE 1
SSP ANY ;WAS POINTING BACK AT STACK POINTER
;SAVED EARLIER, NOW AT 2
WRIT 0 ;SHOULD WRITE 2
ADJ ANY,0
WRITC 0 ;SHOULD WRITE 1
DATA &HFF
DATA &HFF
ADDC
ADJ EQ, 0 ;ADJUST SHOULD FAIL
WRITC 0 ;SHOULD WRITE FE
ADJ NE, 0 ;ADJUST SHOULD SUCCEED
WRIT 0 ;SHOULD WRITE FF
WRITC 0 ;SHOULD WRITE 1
ADJ NEG, 0 ;ADJUST SHOULD SUCCEED
WRIT 0 ;SHOULD WRITE 2
WRITC 0 ;SHOULD WRITE FE
ADJ CY, 0 ;ADJUST SHOULD SUCCEED
WRITC 0 ;SHOULD WRITE FF,
ADJ OV, 0 ;ADJ SHOULD FAIL
WRITC 0 ;SHOULD WRITE FF
ADJ GE, 0 ;ADJ SHOULD FAIL
WRITC 0 ;SHOULD WRITE FF
ADJ LE, 0 ;ADJUST SHOULD SUCCEED
WRITC 0 ;SHOULD WRITE 1
Appendix E - VHDL Model Test Results

This appendix lists the results of running the Level 0 test program discussed in Appendix D on the VHDL model. The columns in the report represent:

1. Time (ns)
2. IP2PCINIT (handshake out from the processor)
3. PC2IPINIT (handshake out from the program controller)
4. SWBUS (the status word)
5. PORT0IP2PC (processor output port 0)
6. PORT0PC2IP (processor input port 0)

Note that the results are slightly different than those run on the FPGA model because the VHDL model has a larger DS so that wrap-around does not occur at the same point.
<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Distance (m)</th>
<th>Speed (m/s)</th>
<th>Acceleration (m/s²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0</td>
<td>8000000</td>
<td>8000000</td>
</tr>
<tr>
<td>25.0</td>
<td>0</td>
<td>8000000</td>
<td>1</td>
</tr>
<tr>
<td>30.0</td>
<td>1</td>
<td>8000000</td>
<td>1</td>
</tr>
<tr>
<td>40.0</td>
<td>0</td>
<td>8000000</td>
<td>188</td>
</tr>
<tr>
<td>45.0</td>
<td>1</td>
<td>1</td>
<td>188</td>
</tr>
<tr>
<td>50.0</td>
<td>0</td>
<td>1</td>
<td>188</td>
</tr>
<tr>
<td>55.0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>60.0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>70.0</td>
<td>0</td>
<td>1</td>
<td>188</td>
</tr>
<tr>
<td>75.0</td>
<td>1</td>
<td>2</td>
<td>188</td>
</tr>
<tr>
<td>80.0</td>
<td>0</td>
<td>2</td>
<td>188</td>
</tr>
<tr>
<td>85.0</td>
<td>0</td>
<td>2</td>
<td>180</td>
</tr>
<tr>
<td>90.0</td>
<td>0</td>
<td>2</td>
<td>180</td>
</tr>
<tr>
<td>100.0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>105.0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>115.0</td>
<td>0</td>
<td>2</td>
<td>188</td>
</tr>
<tr>
<td>120.0</td>
<td>1</td>
<td>3</td>
<td>188</td>
</tr>
<tr>
<td>125.0</td>
<td>0</td>
<td>3</td>
<td>188</td>
</tr>
<tr>
<td>130.0</td>
<td>0</td>
<td>3</td>
<td>180</td>
</tr>
<tr>
<td>135.0</td>
<td>0</td>
<td>3</td>
<td>180</td>
</tr>
<tr>
<td>145.0</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>150.0</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>160.0</td>
<td>0</td>
<td>3</td>
<td>184</td>
</tr>
<tr>
<td>165.0</td>
<td>1</td>
<td>4</td>
<td>184</td>
</tr>
<tr>
<td>170.0</td>
<td>0</td>
<td>4</td>
<td>184</td>
</tr>
<tr>
<td>175.0</td>
<td>0</td>
<td>4</td>
<td>184</td>
</tr>
<tr>
<td>180.0</td>
<td>1</td>
<td>180</td>
<td>184</td>
</tr>
<tr>
<td>185.0</td>
<td>0</td>
<td>180</td>
<td>184</td>
</tr>
<tr>
<td>190.0</td>
<td>0</td>
<td>180</td>
<td>184</td>
</tr>
<tr>
<td>195.0</td>
<td>1</td>
<td>3</td>
<td>184</td>
</tr>
<tr>
<td>200.0</td>
<td>0</td>
<td>3</td>
<td>184</td>
</tr>
<tr>
<td>205.0</td>
<td>0</td>
<td>3</td>
<td>184</td>
</tr>
<tr>
<td>210.0</td>
<td>1</td>
<td>180</td>
<td>184</td>
</tr>
<tr>
<td>215.0</td>
<td>0</td>
<td>180</td>
<td>184</td>
</tr>
<tr>
<td>220.0</td>
<td>0</td>
<td>180</td>
<td>184</td>
</tr>
<tr>
<td>225.0</td>
<td>1</td>
<td>2</td>
<td>184</td>
</tr>
<tr>
<td>230.0</td>
<td>0</td>
<td>2</td>
<td>184</td>
</tr>
<tr>
<td>235.0</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>240.0</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
250.0 0 0 0 2 1
255.0 0 1 0 2 1
265.0 0 0 0 2 141
270.0 0 1 1 2 141
280.0 0 0 1 2 188
285.0 1 1 1 1 188
290.0 0 1 1 1 188
295.0 0 0 1 1 150
300.0 0 1 0 1 150
310.0 0 0 0 1 188
315.0 1 1 0 2 188
320.0 0 0 0 2 188
325.0 0 0 0 0 2 0
330.0 0 0 1 0 2 0
340.0 0 0 0 0 2 143
345.0 0 0 1 1 2 143
355.0 0 0 0 1 2 184
360.0 1 1 1 0 184
365.0 0 0 1 1 0 184
370.0 0 0 0 1 0 FF
375.0 0 0 1 1 0 FF
385.0 0 0 0 1 0 142
390.0 0 0 1 0 0 142
400.0 0 0 0 0 0 188
405.0 1 1 0 2 188
410.0 0 0 1 0 2 188
415.0 0 0 0 0 2 0
420.0 0 0 1 0 2 0
430.0 0 0 0 0 2 0
435.0 0 0 1 0 2 0
445.0 0 0 0 0 2 144
450.0 0 0 1 1 2 144
460.0 0 0 0 1 2 188
465.0 1 1 1 0 188
470.0 0 0 1 1 0 188
475.0 0 0 0 1 0 145
480.0 0 0 1 1 0 145
490.0 0 0 0 1 0 188
495.0 1 1 1 0 188
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>500.0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>505.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>FF</td>
<td></td>
</tr>
<tr>
<td>510.0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>FF</td>
<td></td>
</tr>
<tr>
<td>520.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>525.0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>535.0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>540.0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>FF</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>545.0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>FF</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>550.0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>FF</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>555.0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>FF</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>565.0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>FF</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>570.0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>FF</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>575.0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>FF</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>580.0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>FF</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>585.0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>FF</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>595.0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>FF</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>600.0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>FF</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>610.0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>FF</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>615.0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>54</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>620.0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>54</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>625.0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>54</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>630.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>54</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>640.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>54</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>645.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2A</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>650.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2A</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>655.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2A</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>660.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2A</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>670.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2A</td>
<td>14A</td>
<td></td>
</tr>
<tr>
<td>675.0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>2A</td>
<td>14A</td>
<td></td>
</tr>
<tr>
<td>685.0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>2A</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>690.0</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>54</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>695.0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>54</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>700.0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>54</td>
<td>14B</td>
<td></td>
</tr>
<tr>
<td>705.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>54</td>
<td>14B</td>
<td></td>
</tr>
<tr>
<td>715.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>54</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>720.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2A</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>725.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2A</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>730.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2A</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>735.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2A</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>Value</td>
<td>Type</td>
<td>Code</td>
<td>Result</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>--------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>990.0</td>
<td>1</td>
<td>0</td>
<td>C</td>
<td>161</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000.0</td>
<td>0</td>
<td>0</td>
<td>C</td>
<td>160</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1005.0</td>
<td>0</td>
<td>1</td>
<td>C</td>
<td>160</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1015.0</td>
<td>0</td>
<td>0</td>
<td>C</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1020.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1025.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1030.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1035.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>164</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1045.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1050.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1055.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1060.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1065.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1075.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>165</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1080.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>165</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1090.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>164</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1095.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>164</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1105.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1115.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1120.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1125.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>162</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1135.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1140.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1145.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1150.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1155.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1165.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1170.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1175.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1180.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1185.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1195.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1200.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1210.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1215.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1225.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1230.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1235.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>Status</td>
<td>Counter</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1240</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1245</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1250</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1260</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1265</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1270</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1275</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1285</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1290</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1295</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1300</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1305</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1315</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1320</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1330</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1335</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1345</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1350</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1360</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1365</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1370</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1375</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1380</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1385</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1390</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1395</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1405</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1410</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1420</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1425</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1430</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1435</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1440</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1450</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1455</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1460</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1465</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1710.0 1 1 C 102 188
1715.0 0 1 C 102 188
1720.0 0 0 C 102 1C7
1725.0 0 1 C 102 1C7
1735.0 0 0 C 102 188
1740.0 1 1 C 102 188
1745.0 0 1 C 102 188
1750.0 0 0 C 102 188

Time(ns) ^/ip2pc0init ^/port0ip2pc
 ^/pc2ip0init ^/port0pc2ip
 ^/swbus
The second test run on the VHDL model was to verify that instructions could be loaded executed from the stack. The following short program loads a subprogram which, when executed, increments and writes the number on the top of the DS. Note that the subprogram must load in the reverse of execution order. The results of executing this program are shown in the timing diagram below.

LOAD 166 ; BEGIN LOADING PROGRAM
HALT 170 ; END OF SUBPROGRAM
WRITC 188 ; WRITE TOS BUT PRESERVE IT
ADD 150 ; ADD THE TWO NUMBERS
INST 1 101 ; CONSTANT '1' ENCODED AS AN
; INSTRUCTION TO GET IT ON TO
; THE IS
PUSH 190 ; PUSH THE NEXT IS NUMBER ON TO
; THE TOP OF THE DS.
RESUME 167 ; RESUME NORMAL MODE
DATA 0 0 ; PUSH A '0' ON TO THE DS
EXEC ANY 178 ; EXECUTE THE SUBPROGRAM
END FFFF ; DONE

```
LOAD 166 ; BEGIN LOADING PROGRAM
HALT 170 ; END OF SUBPROGRAM
WRITC 188 ; WRITE TOS BUT PRESERVE IT
ADD 150 ; ADD THE TWO NUMBERS
INST 1 101 ; CONSTANT '1' ENCODED AS AN
; INSTRUCTION TO GET IT ON TO
; THE IS
PUSH 190 ; PUSH THE NEXT IS NUMBER ON TO
; THE TOP OF THE DS.
RESUME 167 ; RESUME NORMAL MODE
DATA 0 0 ; PUSH A '0' ON TO THE DS
EXEC ANY 178 ; EXECUTE THE SUBPROGRAM
END FFFF ; DONE
```
Appendix F - FPGA Logic Diagrams

This appendix contains the logic diagrams used in the generation of the final FPGA-based processor. These were created in Mentor Graphics NETED, v7.

There are eight diagrams:

1. Top Level Schematic
2. ALU Bit
3. Arithmetic/Logic Unit
4. Input/Output Unit
5. Data Stack
6. Control Register Block
7. Clock Phase Generator
8. Control Unit
Schematic F-4
Input/Output Unit
IXAip108
Matthew Dinmore
September, 1993

[Diagram of an input/output unit with various labels and connections, including DFCI, CLA, and input/output ports.]
Schematic F-6
Control Register Block
IXAip108
Matthew Dinmore
September, 1993
Schematic F-7
Clock Phase Generator
IXAip108
Matthew Dinmore
September, 1993
Appendix G - Test Results for the FPGA System

This appendix contains the simulation results for the Level 0 test program run on the FPGA model. It was run against the back-annotated model using maximum delays.

The simulation results have been annotated along the right-hand side to make them easier to follow.
<table>
<thead>
<tr>
<th>TIME</th>
<th>^inbus</th>
<th>^SWs</th>
<th>^SWc</th>
<th>^SWz</th>
<th>^IHo</th>
<th>^OHo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>^outbus</td>
<td>^SWn</td>
<td>^SWv</td>
<td>^IHn</td>
<td>^OHn</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>000z</td>
<td>000</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1z</td>
</tr>
<tr>
<td>55.2</td>
<td>000z</td>
<td>000</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1z</td>
</tr>
<tr>
<td>55.8</td>
<td>000z</td>
<td>000</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1z</td>
</tr>
<tr>
<td>57.2</td>
<td>000z</td>
<td>000</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>58.0</td>
<td>000z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>68.9</td>
<td>000z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>5000.0</td>
<td>000z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>6000.0</td>
<td>000z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>1064.8</td>
<td>000z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>11000.0</td>
<td>000z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>11047.6</td>
<td>000z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>12000.0</td>
<td>001z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>16064.8</td>
<td>001z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>17000.0</td>
<td>001z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>17047.6</td>
<td>001z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>18000.0</td>
<td>188z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>20603.4</td>
<td>188z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>23000.0</td>
<td>188z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>23047.6</td>
<td>188z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>23070.3</td>
<td>188z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>23078.0</td>
<td>188z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>25000.0</td>
<td>188z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>25046.0</td>
<td>188z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>2600.0</td>
<td>188z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>27000.0</td>
<td>002z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>31064.8</td>
<td>002z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>32000.0</td>
<td>002z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>32047.6</td>
<td>002z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>33000.0</td>
<td>188z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>35603.4</td>
<td>188z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>38000.0</td>
<td>188z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>38047.6</td>
<td>188z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>38070.8</td>
<td>188z</td>
<td>003</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>38071.4</td>
<td>188z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>38078.0</td>
<td>188z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>40000.0</td>
<td>188z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>40046.0</td>
<td>188z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>41000.0</td>
<td>188z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>42000.0</td>
<td>180z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>44603.4</td>
<td>180z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>47000.0</td>
<td>180z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>47047.6</td>
<td>180z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>48000.0</td>
<td>003z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>48113.9</td>
<td>003z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>53000.0</td>
<td>003z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>53047.6</td>
<td>003z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>54000.0</td>
<td>188z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>56603.4</td>
<td>188z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
</tr>
<tr>
<td>59000.0</td>
<td>188z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>59047.6</td>
<td>188z</td>
<td>002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>59070.3</td>
<td>188z</td>
<td>003</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>59078.0</td>
<td>188z</td>
<td>003</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
<tr>
<td>61000.0</td>
<td>188z</td>
<td>003</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
</tr>
</tbody>
</table>

processor reset
; load a '1'
; write a '1'
; load a '2'
; write a '2'
; read a...
; '3'
; write a '3'
<table>
<thead>
<tr>
<th>TIME</th>
<th>^inbus</th>
<th>^SWs</th>
<th>^Swc</th>
<th>^Swz</th>
<th>^IHo</th>
<th>^OHo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>^outbus</td>
<td>^SWn</td>
<td>^SWv</td>
<td>^IHi</td>
<td>^OHo</td>
<td></td>
</tr>
<tr>
<td>116000.0</td>
<td>184z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>116047.6</td>
<td>184z</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>116071.4</td>
<td>184z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>116071.5</td>
<td>184z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>116078.0</td>
<td>184z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>118000.0</td>
<td>184z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>118046.0</td>
<td>184z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>119000.0</td>
<td>184z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>120000.0</td>
<td>001z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>124064.8</td>
<td>001z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>125000.0</td>
<td>001z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>125047.6</td>
<td>001z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>126000.0</td>
<td>001z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>130064.8</td>
<td>001z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>131000.0</td>
<td>001z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>131047.6</td>
<td>001z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>132000.0</td>
<td>141z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>134603.4</td>
<td>141z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>137000.0</td>
<td>141z</td>
<td>004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>137047.6</td>
<td>141z</td>
<td>004</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>137088.4</td>
<td>141z</td>
<td>004</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>138000.0</td>
<td>188z</td>
<td>004</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>140603.4</td>
<td>188z</td>
<td>004</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>143000.0</td>
<td>188z</td>
<td>004</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>143047.6</td>
<td>188z</td>
<td>004</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>143069.8</td>
<td>188z</td>
<td>00C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>143070.4</td>
<td>188z</td>
<td>04C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>143070.8</td>
<td>188z</td>
<td>04E</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>143071.1</td>
<td>188z</td>
<td>05E</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>143071.5</td>
<td>188z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>143078.0</td>
<td>188z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>145000.0</td>
<td>188z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>145046.0</td>
<td>188z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>146000.0</td>
<td>188z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>147000.0</td>
<td>150z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>149603.4</td>
<td>150z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>152000.0</td>
<td>150z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>152047.6</td>
<td>150z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>153000.0</td>
<td>188z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>155603.4</td>
<td>188z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>158000.0</td>
<td>188z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>158047.6</td>
<td>188z</td>
<td>0FE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>158070.3</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>158078.0</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>160000.0</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>160046.0</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>161000.0</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>162000.0</td>
<td>000z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>166064.8</td>
<td>000z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>167000.0</td>
<td>000z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>167047.6</td>
<td>000z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>168000.0</td>
<td>143z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>170603.4</td>
<td>143z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

;write a '4'
;load a '1'
;invert it
;add '1' + 'FE'
;write 'FF'
;load a '0'
;ANDC it
<table>
<thead>
<tr>
<th>TIME</th>
<th>^inbus</th>
<th>^SWs</th>
<th>^Swc</th>
<th>^SWz</th>
<th>^IHo</th>
<th>^OHo</th>
</tr>
</thead>
<tbody>
<tr>
<td>173000.0</td>
<td>143z 0FF</td>
<td>0 1 0 0 0 0z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>173047.6</td>
<td>143z 0FF</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>173077.5</td>
<td>143z 0FF</td>
<td>0 1 0 0 1 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>173092.7</td>
<td>143z 0FF</td>
<td>0 0 0 0 0 1 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>174000.0</td>
<td>184z 0FF</td>
<td>0 0 0 0 1 1z 0 0z 0</td>
<td>;write a '0'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>176603.4</td>
<td>184z 0FF</td>
<td>0 0 0 0 1 1z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>179000.0</td>
<td>184z 0FF</td>
<td>0 0 0 0 1 0z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>179047.6</td>
<td>184z 0FF</td>
<td>0 0 0 0 1 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>179071.3</td>
<td>184z OB7</td>
<td>0 0 0 0 1 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>179071.4</td>
<td>184z OB6</td>
<td>0 0 0 0 1 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>179071.9</td>
<td>184z OA4</td>
<td>0 0 0 0 1 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>179072.5</td>
<td>184z 024</td>
<td>0 0 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>180000.0</td>
<td>184z 000</td>
<td>0 0 0 0 0 1 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>182000.0</td>
<td>184z 000</td>
<td>0 0 0 0 0 1 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>183000.0</td>
<td>0FFz 000</td>
<td>0 0 0 0 0 1 1z 0 0z 0</td>
<td>;load an 'FF'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>187064.8</td>
<td>0FFz 000</td>
<td>0 0 0 0 0 1 1z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>188000.0</td>
<td>0FFz 000</td>
<td>0 0 0 0 0 1 0z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>188047.6</td>
<td>0FFz 000</td>
<td>0 0 0 0 0 1 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>189000.0</td>
<td>142z 000</td>
<td>0 0 0 0 0 1 1z 0 0z 0</td>
<td>;AND it</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>191603.4</td>
<td>142z 000</td>
<td>0 0 0 0 0 1 1z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>194000.0</td>
<td>142z 000</td>
<td>0 0 0 0 0 1 0z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>194047.6</td>
<td>142z 000</td>
<td>0 0 0 0 0 1 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>194079.5</td>
<td>142z 000</td>
<td>0 0 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>194088.4</td>
<td>142z 000</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>195000.0</td>
<td>188z 000</td>
<td>0 1 0 0 0 1z 0 0z 0</td>
<td>;write an 'FF'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>197603.4</td>
<td>188z 000</td>
<td>0 1 0 0 0 1z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200000.0</td>
<td>188z 000</td>
<td>0 1 0 0 0 0z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200047.6</td>
<td>188z 000</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200069.8</td>
<td>188z 008</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200070.3</td>
<td>188z 009</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200070.4</td>
<td>188z 049</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200070.8</td>
<td>188z 04B</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200071.1</td>
<td>188z 05B</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200071.5</td>
<td>188z 0FF</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200078.0</td>
<td>188z 0FF</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>202000.0</td>
<td>188z 0FF</td>
<td>0 1 0 0 0 0z 0 1z 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>202046.0</td>
<td>188z 0FF</td>
<td>0 1 0 0 0 0z 0 1z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>203000.0</td>
<td>188z 0FF</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>204000.0</td>
<td>000z 0FF</td>
<td>0 1 0 0 0 1z 0 0z 0</td>
<td>;load a '0'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>208064.8</td>
<td>000z 0FF</td>
<td>0 1 0 0 0 1z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>209000.0</td>
<td>000z 0FF</td>
<td>0 1 0 0 0 0z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>209047.6</td>
<td>000z 0FF</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>210000.0</td>
<td>000z 0FF</td>
<td>0 1 0 0 0 1z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>214064.8</td>
<td>000z 0FF</td>
<td>0 1 0 0 0 1z 1 0z 0</td>
<td>;load a '0'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>215000.0</td>
<td>000z 0FF</td>
<td>0 1 0 0 0 0z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>215047.6</td>
<td>000z 0FF</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>216000.0</td>
<td>144z 0FF</td>
<td>0 1 0 0 0 1z 0 0z 0</td>
<td>;OR them</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>218603.4</td>
<td>144z 0FF</td>
<td>0 1 0 0 0 1z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>221000.0</td>
<td>144z 0FF</td>
<td>0 1 0 0 0 0z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>221047.6</td>
<td>144z 0FF</td>
<td>0 1 0 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>^inbus</td>
<td>^SWs</td>
<td>^SWc</td>
<td>^SWz</td>
<td>^IHo</td>
<td>^OHo</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>221077.5</td>
<td>144z</td>
<td>OFF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>221092.7</td>
<td>144z</td>
<td>OFF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>222000.0</td>
<td>188z</td>
<td>OFF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>224603.4</td>
<td>188z</td>
<td>OFF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>227000.0</td>
<td>188z</td>
<td>OFF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>227047.6</td>
<td>188z</td>
<td>OFF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>227071.3</td>
<td>188z</td>
<td>OB7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>227071.4</td>
<td>188z</td>
<td>OB6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>227071.9</td>
<td>188z</td>
<td>OA4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>227072.5</td>
<td>188z</td>
<td>024</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>227073.2</td>
<td>188z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>227078.0</td>
<td>188z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>229000.0</td>
<td>188z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>230000.0</td>
<td>188z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>231000.0</td>
<td>145z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>233603.4</td>
<td>145z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>236000.0</td>
<td>145z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>236047.6</td>
<td>145z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>236079.5</td>
<td>145z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>238088.4</td>
<td>145z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>237000.0</td>
<td>188z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>239603.4</td>
<td>188z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>242000.0</td>
<td>188z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>242047.6</td>
<td>188z</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>242069.8</td>
<td>188z</td>
<td>008</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>242070.3</td>
<td>188z</td>
<td>009</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>242070.4</td>
<td>188z</td>
<td>049</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>242070.8</td>
<td>188z</td>
<td>04B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>242071.1</td>
<td>188z</td>
<td>05B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>242071.5</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>242078.0</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>244000.0</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>244046.0</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>245000.0</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>246000.0</td>
<td>0FFz</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>250064.8</td>
<td>0FFz</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>251000.0</td>
<td>0FFz</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>251047.6</td>
<td>0FFz</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>252000.0</td>
<td>147z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>254603.4</td>
<td>147z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>257000.0</td>
<td>147z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>257047.6</td>
<td>147z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>257077.5</td>
<td>147z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>257092.7</td>
<td>147z</td>
<td>0FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>258000.0</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>260603.4</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>262000.0</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>263004.7</td>
<td>188z</td>
<td>0FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>263071.3</td>
<td>188z</td>
<td>OB7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>263071.4</td>
<td>188z</td>
<td>OB6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>263071.9</td>
<td>188z</td>
<td>OA4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>263072.5</td>
<td>188z</td>
<td>024</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
TIME  ^inbus  ^SWs  ^SWc  ^SWz  ^IHo  ^OHo
^outbus  ^SWn  ^SWv  ^IHi  ^OHi

263073.2 188z 000 0 0 0 0 1 0z 0 0z 0
263078.0 188z 000 0 0 0 0 1 0z 0 0z 1
265000.0 188z 000 0 0 0 0 1 0z 0 1z 1
265046.0 188z 000 0 0 0 0 1 0z 0 1z 0
266000.0 188z 000 0 0 0 0 1 0z 0 0z 0
267000.0 146z 000 0 0 0 0 1 1z 0 0z 0 ;XOR it
269603.4 146z 000 0 0 0 0 1 1z 1 0z 0
272000.0 146z 000 0 0 0 0 1 0z 1 0z 0
272047.6 146z 000 0 0 0 0 1 0z 0 0z 0
272079.5 146z 000 0 0 0 0 0 0z 0 0z 0
272088.4 146z 000 0 1 0 0 0 0z 0 0z 0
273000.0 188z 000 0 1 0 0 0 1z 0 0z 0 ;write an ‘FF’
275603.4 188z 000 0 1 0 0 0 1z 1 0z 0
278000.0 188z 000 0 1 0 0 0 0z 1 0z 0
278047.6 188z 000 0 1 0 0 0 0z 0 0z 0
278069.8 188z 008 0 1 0 0 0 0z 0 0z 0
278070.3 188z 009 0 1 0 0 0 0z 0 0z 0
278070.4 188z 049 0 1 0 0 0 0z 0 0z 0
278070.8 188z 04B 0 1 0 0 0 0z 0 0z 0
278071.1 188z 05B 0 1 0 0 0 0z 0 0z 0
278071.5 188z 0FF 0 1 0 0 0 0z 0 0z 0
278078.0 188z 0FF 0 1 0 0 0 0z 0 0z 1
280000.0 188z 0FF 0 1 0 0 0 0z 0 1z 1
280046.0 188z 0FF 0 1 0 0 0 0z 0 1z 0
281000.0 188z 0FF 0 1 0 0 0 0z 0 0z 0
282000.0 0AAz 0FF 0 1 0 0 0 1z 0 0z 0 ;load an ‘AA’
286064.8 0AAz 0FF 0 1 0 0 0 1z 1 0z 0
287000.0 0AAz 0FF 0 1 0 0 0 0z 1 0z 0
287047.6 0AAz 0FF 0 1 0 0 0 0z 0 0z 0
288000.0 148z 0FF 0 1 0 0 0 1z 0 0z 0 ;SLL it
290603.4 148z 0FF 0 1 0 0 0 1z 1 0z 0
293000.0 148z 0FF 0 1 0 0 0 0z 1 0z 0
293047.6 148z 0FF 0 1 0 0 0 0z 0 0z 0
293075.9 148z 0FF 0 1 0 0 0 0z 0 0z 0
293092.7 148z 0FF 0 1 0 0 0 0z 0 0z 0
294000.0 188z 0FF 0 0 1 0 0 1z 0 0z 0 ;write a ‘54’
296603.4 188z 0FF 0 0 1 0 0 1z 1 0z 0
299000.0 188z 0FF 0 0 1 0 0 0z 1 0z 0
299047.6 188z 0FF 0 0 1 0 0 0z 0 0z 0
299071.3 188z 0FF 0 0 1 0 0 0z 0 0z 0
299071.4 188z 0FF 0 0 1 0 0 0z 0 0z 0
299072.5 188z 074 0 0 1 0 0 0z 0 0z 0
299073.2 188z 054 0 0 1 0 0 0z 0 0z 0
299078.0 188z 054 0 0 1 0 0 0z 0 0z 1
301000.0 188z 054 0 0 1 0 0 0z 0 1z 1
301046.0 188z 054 0 0 1 0 0 0z 0 1z 0
302000.0 188z 054 0 0 1 0 0 0z 0 0z 0
303000.0 149z 054 0 0 1 0 0 1z 0 0z 0 ;SRL it
305603.4 149z 054 0 0 1 0 0 1z 1 0z 0
308000.0 149z 054 0 0 1 0 0 0z 1 0z 0
308047.6 149z 054 0 0 1 0 0 0z 0 0z 0
308077.5 149z 054 0 0 0 0 0 0z 0 0z 0
<table>
<thead>
<tr>
<th>TIME</th>
<th>^inbus</th>
<th>^SWs</th>
<th>^SWc</th>
<th>^SWz</th>
<th>^IHo</th>
<th>^OHo</th>
<th>^outbus</th>
<th>^SWn</th>
<th>^SWv</th>
<th>^Hi</th>
<th>^Ohi</th>
</tr>
</thead>
<tbody>
<tr>
<td>309000.0</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>;write a '2A'</td>
</tr>
<tr>
<td>311603.4</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>314000.0</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>314047.6</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>314069.8</td>
<td>188z 05C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>314070.8</td>
<td>188z 05E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>314071.3</td>
<td>188z 01E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>314071.5</td>
<td>188z 03E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>314071.9</td>
<td>188z 02E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>314073.2</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>314078.0</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>316000.0</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>316046.0</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>317000.0</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>318000.0</td>
<td>0AAz 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>;load an 'AA'</td>
</tr>
<tr>
<td>322064.8</td>
<td>0AAz 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>323000.0</td>
<td>0AAz 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>323047.6</td>
<td>0AAz 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>324000.0</td>
<td>14Az 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>;SLA it</td>
</tr>
<tr>
<td>326603.4</td>
<td>14Az 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>329000.0</td>
<td>14Az 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>329047.6</td>
<td>14Az 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>329075.9</td>
<td>14Az 02A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>329078.1</td>
<td>14Az 02A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>330000.0</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>;write a '54'</td>
</tr>
<tr>
<td>332603.4</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>335000.0</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>335047.6</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>335070.4</td>
<td>188z 06A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>335071.1</td>
<td>188z 07A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>335071.3</td>
<td>188z 072</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>335071.5</td>
<td>188z 076</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>335071.9</td>
<td>188z 074</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>335073.2</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>335078.0</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>337000.0</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>337046.0</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>338000.0</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>339000.0</td>
<td>14Bz 054</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>;SRA it</td>
</tr>
<tr>
<td>341603.4</td>
<td>14Bz 054</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>344000.0</td>
<td>14Bz 054</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>344047.6</td>
<td>14Bz 054</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>344077.5</td>
<td>14Bz 054</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>344080.3</td>
<td>14Bz 054</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>345000.0</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>;write a '2A'</td>
</tr>
<tr>
<td>347603.4</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>350000.0</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>350047.6</td>
<td>188z 054</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>350069.8</td>
<td>188z 05C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>350070.8</td>
<td>188z 05E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>350071.3</td>
<td>188z 01E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>350071.5</td>
<td>188z 03E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>350071.9</td>
<td>188z 02E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>^inbus</td>
<td>^SWs</td>
<td>^SWc</td>
<td>^SWz</td>
<td>^IHo</td>
<td>^OHo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>^outbus</td>
<td>^SWn</td>
<td>^SWv</td>
<td>^IHi</td>
<td>^OHi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>350073.2</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>350078.0</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>352000.0</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>352046.0</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>353000.0</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>354000.0</td>
<td>OA Az 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>358064.8</td>
<td>OA Az 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>1z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>359000.0</td>
<td>OA Az 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>359047.6</td>
<td>OA Az 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>360000.0</td>
<td>14Cz 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>362603.4</td>
<td>14Cz 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>365000.0</td>
<td>14Cz 02A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>1z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>365047.6</td>
<td>14Cz 02A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>365075.9</td>
<td>14Cz 02A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>366000.0</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>368603.4</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>371000.0</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>371047.6</td>
<td>188z 02A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>371070.3</td>
<td>188z 02B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>371070.4</td>
<td>188z 02B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>371071.1</td>
<td>188z 02B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>371071.3</td>
<td>188z 073</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>371071.5</td>
<td>188z 077</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>371071.9</td>
<td>188z 075</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>371073.2</td>
<td>188z 055</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>371078.0</td>
<td>188z 055</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>373000.0</td>
<td>188z 055</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>373046.0</td>
<td>188z 055</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>374000.0</td>
<td>188z 055</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>375000.0</td>
<td>14Dz 055</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>377603.4</td>
<td>14Dz 055</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>380000.0</td>
<td>14Dz 055</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>380047.6</td>
<td>14Dz 055</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>380088.4</td>
<td>14Dz 055</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>381000.0</td>
<td>188z 055</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>383603.4</td>
<td>188z 055</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386000.0</td>
<td>188z 055</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386047.6</td>
<td>188z 055</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386069.8</td>
<td>188z 05D</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386070.8</td>
<td>188z 05F</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386071.3</td>
<td>188z 01F</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386071.4</td>
<td>188z 01E</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386071.5</td>
<td>188z 0BE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386071.9</td>
<td>188z 0AE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386073.2</td>
<td>188z 0AA</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>386078.0</td>
<td>188z 0AA</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>388000.0</td>
<td>188z 0AA</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>388046.0</td>
<td>188z 0AA</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>390000.0</td>
<td>188z 0AA</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>392603.4</td>
<td>14Dz 0AA</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>395000.0</td>
<td>14Dz 0AA</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>395047.6</td>
<td>14Dz 0AA</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

;load an ‘AA’
;SLR it
;write a ‘55’
;SPR it
;write ‘AA’
;SRR it
TIME  inbus  ^SWs  ^Swc  ^SWz  ^IHo  ^OHo
      ^outbus  ^Swn  ^SWv  ^IHi  ^OHi
395077.5  14Dz  0AA  0  1  0  0  0  0z  0  0  0z  0
395092.7  14Dz  0AA  0  0  0  0  0  0z  0  0  0z  0
396000.0  188z  0AA  0  0  0  0  0  1z  0  0  0z  0  ;write '55'
398603.4  188z  0AA  0  0  0  0  0  1z  1  0z  0
401000.0  188z  0AA  0  0  0  0  0  0z  1  0z  0
401047.6  188z  0AA  0  0  0  0  0  0z  0  0  0z  0
401070.3  188z  0AB  0  0  0  0  0  0z  0  0  0z  0
401070.4  188z  0EB  0  0  0  0  0  0z  0  0  0z  0
401071.1  188z  0FB  0  0  0  0  0  0z  0  0  0z  0
401071.3  188z  0F3  0  0  0  0  0  0z  0  0  0z  0
401071.5  188z  0F7  0  0  0  0  0  0z  0  0  0z  0
401071.9  188z  0F5  0  0  0  0  0  0z  0  0  0z  0
401072.5  188z  075  0  0  0  0  0  0z  0  0  0z  0
401073.2  188z  055  0  0  0  0  0  0z  0  0  0z  0
401078.0  188z  055  0  0  0  0  0  0z  0  0z  1
403000.0  188z  055  0  0  0  0  0  0z  0  1z  1
403046.0  188z  055  0  0  0  0  0  0z  0  1z  0
404000.0  188z  055  0  0  0  0  0  0z  0  0z  0
405087.1  188z  015  0  0  0  0  0  0z  0  0  0z  0
405087.2  188z  014  0  0  0  0  0  0z  0  0  0z  0
405087.7  188z  004  0  0  0  0  0  0z  0  0  0z  0
405089.0  188z  000  0  0  0  0  0  0z  0  0  0z  0
407000.0  160z  000  0  0  0  0  0  1z  0  0z  0  ;PSW
4099603.4  160z  000  0  0  0  0  0  1z  1  0z  0
412000.0  160z  000  0  0  0  0  0  0z  1  0z  0
412047.6  160z  000  0  0  0  0  0  0z  0  0  0z  0
413000.0  188z  000  0  0  0  0  0  1z  0  0z  0  ;write '0'
415603.4  188z  000  0  0  0  0  0  1z  1  0z  0
418000.0  188z  000  0  0  0  0  0  0z  1  0z  0
418047.6  188z  000  0  0  0  0  0  0z  0  0  0z  0
418078.0  188z  000  0  0  0  0  0  0z  0  0z  1
420000.0  188z  000  0  0  0  0  0  0z  0  1z  1
420046.0  188z  000  0  0  0  0  0  0z  0  1z  0
421000.0  188z  000  0  0  0  0  0  0z  0  0  0z  0
422000.0  0FFz  000  0  0  0  0  0  1z  0  0z  0  ;load 'FF'
426064.8  0FFz  000  0  0  0  0  0  1z  1  0z  0
427000.0  0FFz  000  0  0  0  0  0  0z  1  0z  0
427047.6  0FFz  000  0  0  0  0  0  0z  0  0  0z  0
428000.0  0FFz  000  0  0  0  0  0  1z  0  0z  0
432064.8  0FFz  000  0  0  0  0  0  1z  1  0z  0  ;load 'FF'
433000.0  0FFz  000  0  0  0  0  0  0z  1  0z  0
433047.6  0FFz  000  0  0  0  0  0  0z  0  0  0z  0
434000.0  150z  000  0  0  0  0  0  1z  0  0z  0  ;ADD
436603.4  150z  000  0  0  0  0  0  1z  1  0z  0
439000.0  150z  000  0  0  0  0  0  0z  1  0z  0
439047.6  150z  000  0  0  0  0  0  0z  0  0  0z  0
439075.9  150z  000  0  0  1  0  0  0z  0  0  0z  0
439088.4  150z  000  0  1  1  0  0  0z  0  0  0z  0
440000.0  160z  000  0  1  1  0  0  1z  0  0z  0  ;PSW
442063.4  160z  000  0  1  1  0  0  1z  1  0z  0
445000.0  160z  000  0  1  1  0  0  0z  1  0z  0
445047.6  160z  000  0  1  1  0  0  0z  0  0  0z  0
446000.0  184z  000  0  1  1  0  0  1z  0  0z  0  ;write 'OC'
<table>
<thead>
<tr>
<th>TIME</th>
<th>^inbus</th>
<th>^SWs</th>
<th>^SWc</th>
<th>^SWz</th>
<th>^IHo</th>
<th>^OHo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>^outbus</td>
<td>^SWn</td>
<td>^SWv</td>
<td>^IHc</td>
<td>^OHc</td>
<td></td>
</tr>
<tr>
<td>448603.4</td>
<td>184z 000</td>
<td>0 1 1 0 0 1z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>451000.0</td>
<td>184z 000</td>
<td>0 1 1 0 0 0z 1 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>451047.6</td>
<td>184z 000</td>
<td>0 1 1 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>451069.8</td>
<td>184z 008</td>
<td>0 1 1 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>451071.5</td>
<td>184z 00C</td>
<td>0 1 1 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>451078.0</td>
<td>184z 00C</td>
<td>0 1 1 0 0 0z 0 0z 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>453000.0</td>
<td>184z 00C</td>
<td>0 1 1 0 0 0z 0 1z 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>453046.0</td>
<td>184z 00C</td>
<td>0 1 1 0 0 0z 0 1z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>454000.0</td>
<td>184z 00C</td>
<td>0 1 1 0 0 0z 0 0z 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 455000.0 | 000z 00C | 0 1 1 0 0 1z 0 0z 0 | ;load '0'
| 459064.8 | 000z 00C | 0 1 1 0 0 1z 1 0z 0 |
| 460000.0 | 000z 00C | 0 1 1 0 0 0z 1 0z 0 |
| 460047.6 | 000z 00C | 0 1 1 0 0 0z 0 0z 0 |
| 461000.0 | 161z 00C | 0 1 1 0 0 1z 0 0z 0 | ;SSW
| 463603.4 | 161z 00C | 0 1 1 0 0 1z 1 0z 0 |
| 466000.0 | 161z 00C | 0 1 1 0 0 0z 1 0z 0 |
| 466047.6 | 161z 00C | 0 1 1 0 0 0z 0 0z 0 |
| 466077.5 | 161z 00C | 0 1 0 0 0 0z 0 0z 0 |
| 466092.7 | 161z 00C | 0 0 0 0 0 0z 0 0z 0 |
| 467000.0 | 160z 00C | 0 0 0 0 0 1z 0 0z 0 | ;PSW
| 469603.4 | 160z 00C | 0 0 0 0 0 1z 1 0z 0 |
| 472000.0 | 160z 00C | 0 0 0 0 0 0z 1 0z 0 |
| 472047.6 | 160z 00C | 0 0 0 0 0 0z 0 0z 0 |
| 473000.0 | 184z 00C | 0 0 0 0 0 1z 0 0z 0 | ;write '0'
| 475603.4 | 184z 00C | 0 0 0 0 0 1z 1 0z 0 |
| 478000.0 | 184z 00C | 0 0 0 0 0 0z 1 0z 0 |
| 478047.6 | 184z 00C | 0 0 0 0 0 0z 0 0z 0 |
| 478071.3 | 184z 004 | 0 0 0 0 0 0z 0 0z 0 |
| 478073.2 | 184z 000 | 0 0 0 0 0 0z 0 0z 0 |
| 478078.0 | 184z 000 | 0 0 0 0 0 0z 0 0z 1 |
| 480000.0 | 184z 000 | 0 0 0 0 0 0z 0 1z 1 |
| 480046.0 | 184z 000 | 0 0 0 0 0 0z 0 1z 0 |
| 481000.0 | 184z 000 | 0 0 0 0 0 0z 0 0z 0 |
| 482000.0 | 164z 000 | 0 0 0 0 0 1z 0 0z 0 | ;PIO
| 484603.4 | 164z 000 | 0 0 0 0 0 1z 1 0z 0 |
| 487000.0 | 164z 000 | 0 0 0 0 0 0z 1 0z 0 |
| 487047.6 | 164z 000 | 0 0 0 0 0 0z 0 0z 0 | ;write 0
| 488000.0 | 184z 000 | 0 0 0 0 0 1z 0 0z 0 |
| 490603.4 | 184z 000 | 0 0 0 0 0 1z 1 0z 0 |
| 493000.0 | 184z 000 | 0 0 0 0 0 0z 1 0z 0 |
| 493047.6 | 184z 000 | 0 0 0 0 0 0z 0 0z 0 |
| 493078.0 | 184z 000 | 0 0 0 0 0 0z 0 0z 1 |
| 495000.0 | 184z 000 | 0 0 0 0 0 0z 0 1z 1 |
| 495046.0 | 184z 000 | 0 0 0 0 0 0z 0 1z 0 |
| 496000.0 | 184z 000 | 0 0 0 0 0 0z 0 0z 0 |
| 497000.0 | 001z 000 | 0 0 0 0 0 1z 0 0z 0 | ;load '1'
| 500164.8 | 001z 000 | 0 0 0 0 0 1z 1 0z 0 |
| 502000.0 | 001z 000 | 0 0 0 0 0 0z 1 0z 0 |
| 502047.6 | 001z 000 | 0 0 0 0 0 0z 0 0z 0 |
| 503000.0 | 165z 000 | 0 0 0 0 0 1z 0 0z 0 | ;SIO
<p>| 505603.4 | 165z 000 | 0 0 0 0 0 1z 1 0z 0 |
| 508000.0 | 165z 000 | 0 0 0 0 0 0z 1 0z 0 |
| 508047.6 | 165z 000 | 0 0 0 0 0 0z 0 0z 0 |</p>
<table>
<thead>
<tr>
<th>TIME</th>
<th>^inbus</th>
<th>^SWs</th>
<th>^SWc</th>
<th>^SWz</th>
<th>^IHo</th>
<th>^OHo</th>
<th>^outbus</th>
<th>^SWn</th>
<th>^SWv</th>
<th>^IH1</th>
<th>^OH1</th>
</tr>
</thead>
<tbody>
<tr>
<td>508073.7</td>
<td>165z</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>509000.0</td>
<td>164z</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>511603.4</td>
<td>164z</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>514000.0</td>
<td>164z</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>514047.6</td>
<td>164z</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>515000.0</td>
<td>184z</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>517603.4</td>
<td>184z</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>520000.0</td>
<td>184z</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>520047.6</td>
<td>184z</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>520707.3</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>52078.0</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>1</td>
</tr>
<tr>
<td>522000.0</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>1</td>
</tr>
<tr>
<td>522046.0</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>0</td>
</tr>
<tr>
<td>523000.0</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>524000.0</td>
<td>162z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>526603.4</td>
<td>162z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>529000.0</td>
<td>162z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>529047.6</td>
<td>162z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>530000.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>532603.4</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>535000.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>535047.6</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>535070.8</td>
<td>188z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>535071.4</td>
<td>188z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>535078.0</td>
<td>188z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>1</td>
</tr>
<tr>
<td>537000.0</td>
<td>188z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>1</td>
</tr>
<tr>
<td>537046.0</td>
<td>188z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>0</td>
</tr>
<tr>
<td>538000.0</td>
<td>188z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>539000.0</td>
<td>162z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>541603.4</td>
<td>162z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>544000.0</td>
<td>162z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>544047.6</td>
<td>162z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>545000.0</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>547603.4</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>550000.0</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>550047.6</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>550070.3</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>550071.9</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>550078.0</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>1</td>
</tr>
<tr>
<td>552000.0</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>1</td>
</tr>
<tr>
<td>552046.0</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>1z</td>
<td>0</td>
</tr>
<tr>
<td>553000.0</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>554000.0</td>
<td>001z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>558064.8</td>
<td>001z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>559000.0</td>
<td>001z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>559047.6</td>
<td>001z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>560000.0</td>
<td>002z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>564064.8</td>
<td>002z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>565000.0</td>
<td>002z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>565047.6</td>
<td>002z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>566000.0</td>
<td>003z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>0</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>570064.8</td>
<td>003z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>571000.0</td>
<td>003z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td>1</td>
<td>0z</td>
<td>0</td>
</tr>
<tr>
<td>TIME</td>
<td>^inbus</td>
<td>^SWs</td>
<td>^SWc</td>
<td>^SWz</td>
<td>^IHo</td>
<td>^OHo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>571047.6</td>
<td>003z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>572000.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>574603.4</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>577000.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>577047.6</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>577070.8</td>
<td>188z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>577078.0</td>
<td>188z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>579000.0</td>
<td>188z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>579046.0</td>
<td>188z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>580000.0</td>
<td>188z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>581000.0</td>
<td>1A0z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>583603.4</td>
<td>1A0z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>586000.0</td>
<td>1A0z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>588047.6</td>
<td>1A0z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>587000.0</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>589603.4</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>592000.0</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>592047.6</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>592078.0</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>594000.0</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>594046.0</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>595000.0</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>596000.0</td>
<td>1A1z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>598603.4</td>
<td>1A1z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>601000.0</td>
<td>1A1z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>601047.6</td>
<td>1A1z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>602000.0</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>604603.4</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>607000.0</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>607047.6</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>607071.4</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>607078.0</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>609000.0</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>609046.0</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>610000.0</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>611000.0</td>
<td>162z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>613603.4</td>
<td>162z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>616000.0</td>
<td>162z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>616047.6</td>
<td>162z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>617000.0</td>
<td>1C0z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>619603.4</td>
<td>1C0z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>622000.0</td>
<td>1C0z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>622047.6</td>
<td>1C0z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>623000.0</td>
<td>188z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>625603.4</td>
<td>188z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>628000.0</td>
<td>188z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>628047.6</td>
<td>188z</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>628070.3</td>
<td>188z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>628078.0</td>
<td>188z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>630000.0</td>
<td>188z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>630046.0</td>
<td>188z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>631000.0</td>
<td>188z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>632000.0</td>
<td>192z</td>
<td>103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

;write '3'
;GET 0
;write '3'
;GET 1
;write '2'
;PSP
;SSP ANY
;write '3'
;SWAP
<table>
<thead>
<tr>
<th>TIME</th>
<th>inbus</th>
<th>Sws</th>
<th>Swc</th>
<th>Swz</th>
<th>Ih</th>
<th>oHo</th>
</tr>
</thead>
<tbody>
<tr>
<td>634603.4</td>
<td>192z 103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>637000.0</td>
<td>192z 103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>640603.4</td>
<td>184z 103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>652000.0</td>
<td>184z 102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>661000.0</td>
<td>184z 103</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>676000.0</td>
<td>184z 101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>682603.4</td>
<td>1C0z 102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TIME</td>
<td>inbus</td>
<td>Sws</td>
<td>Swc</td>
<td>Swz</td>
<td>IHo</td>
<td>OHo</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>691071.9</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>691078.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>693000.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>693046.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>694000.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>695000.0</td>
<td>0FFz</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>699064.8</td>
<td>0FFz</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>700000.0</td>
<td>0FFz</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>700047.6</td>
<td>0FFz</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>701000.0</td>
<td>0FFz</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>705064.8</td>
<td>0FFz</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>706000.0</td>
<td>0FFz</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>706047.6</td>
<td>0FFz</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>707000.0</td>
<td>151z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>709603.4</td>
<td>151z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>712000.0</td>
<td>151z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>712047.6</td>
<td>151z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>712075.9</td>
<td>151z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>712088.4</td>
<td>151z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>713000.0</td>
<td>1Clz</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>715603.4</td>
<td>1Clz</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>718000.0</td>
<td>1Clz</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>718047.6</td>
<td>1Clz</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>719000.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>721603.4</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>724000.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>724047.6</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>724069.8</td>
<td>188z</td>
<td>109</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>724070.4</td>
<td>188z</td>
<td>149</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>724070.8</td>
<td>188z</td>
<td>14B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>724071.1</td>
<td>188z</td>
<td>15B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>724071.4</td>
<td>188z</td>
<td>15A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>724071.5</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>724078.0</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>726000.0</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>726046.0</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>727000.0</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>728000.0</td>
<td>1C2z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>730603.4</td>
<td>1C2z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>733000.0</td>
<td>1C2z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>733047.6</td>
<td>1C2z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>734000.0</td>
<td>184z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>736003.4</td>
<td>184z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>739000.0</td>
<td>184z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>739047.6</td>
<td>184z</td>
<td>1FE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>739070.3</td>
<td>184z</td>
<td>1FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>739078.0</td>
<td>184z</td>
<td>1FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>741000.0</td>
<td>184z</td>
<td>1FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>741046.0</td>
<td>184z</td>
<td>1FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>742000.0</td>
<td>184z</td>
<td>1FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>743000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>745603.4</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>748000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

; load 'FF'
; ADDC them
; ADJ EQ (fail)
; write 'FE'
; write 'FF'
; write '1'
<table>
<thead>
<tr>
<th>TIME</th>
<th>inbus</th>
<th>SWs</th>
<th>Swc</th>
<th>SWz</th>
<th>IHo</th>
<th>Ho</th>
</tr>
</thead>
<tbody>
<tr>
<td>748047.6</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>748071.3</td>
<td>188z</td>
<td>1B7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>748071.9</td>
<td>188z</td>
<td>1A5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>748072.5</td>
<td>188z</td>
<td>125</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>748073.2</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>748078.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>750000.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>750046.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>751000.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>752000.0</td>
<td>1C3z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>754603.4</td>
<td>1C3z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>757000.0</td>
<td>1C3z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>757047.6</td>
<td>1C3z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>758000.0</td>
<td>1C3z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>760603.4</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>763000.0</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>763047.6</td>
<td>184z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>763070.8</td>
<td>184z</td>
<td>103</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>763071.4</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>763078.0</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>765000.0</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>765046.0</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>766000.0</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>767000.0</td>
<td>188z</td>
<td>102</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>769603.4</td>
<td>184z</td>
<td>102</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>772000.0</td>
<td>188z</td>
<td>102</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>772047.6</td>
<td>188z</td>
<td>102</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>772069.8</td>
<td>188z</td>
<td>10A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>772070.4</td>
<td>188z</td>
<td>14A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>772071.1</td>
<td>188z</td>
<td>15A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>772071.5</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>772078.0</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>774000.0</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>774046.0</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>775000.0</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>776000.0</td>
<td>1C4z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>778603.4</td>
<td>1C4z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>781000.0</td>
<td>1C4z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>781047.6</td>
<td>1C4z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>782000.0</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>784603.4</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>787000.0</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>787047.6</td>
<td>188z</td>
<td>1FE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>787070.3</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>787078.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>789000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>790046.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>791000.0</td>
<td>1C5z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>793603.4</td>
<td>1C5z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>796047.6</td>
<td>1C5z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>797000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

;ADJ NEG (succeed)

;write '2'

;write 'FE'

;ADJ CY (succeed)

;write 'FF'

;write 'FF'

;ADJ OV (fail)
<table>
<thead>
<tr>
<th>TIME</th>
<th>^inbus</th>
<th>^SWs</th>
<th>^Swc</th>
<th>^Swz</th>
<th>^IHo</th>
<th>^OHo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>^outbus</td>
<td>^Swh</td>
<td>^Swv</td>
<td>^IHh</td>
<td>^OHh</td>
<td></td>
</tr>
<tr>
<td>799603.4</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>802000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>802047.6</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>802078.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>804000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>804046.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>805000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>806000.0</td>
<td>1C6z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>808603.4</td>
<td>1C6z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>811000.0</td>
<td>1C6z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>811047.6</td>
<td>1C6z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>812000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>814603.4</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>817000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>817047.6</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>817078.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>819000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>819046.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>820000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>821000.0</td>
<td>1C7z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>823603.4</td>
<td>1C7z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>826000.0</td>
<td>1C7z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>826047.6</td>
<td>1C7z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>827000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>829603.4</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>832000.0</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>832047.6</td>
<td>188z</td>
<td>1FF</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>832071.3</td>
<td>188z</td>
<td>1B7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>832071.9</td>
<td>188z</td>
<td>1A5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>832072.5</td>
<td>188z</td>
<td>125</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>832073.2</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>832078.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>834000.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>834046.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>835000.0</td>
<td>188z</td>
<td>101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

;ADJ GE (fail)

;write 'FF'

;ADJ LE (succeed)

;write '1'
Appendix H - FPGA Place & Route Results

The following pages contain the results of the FPGA place and route process as reported by the Actel Action Logic System software. The list files provided are:

1. The results of automatic pin placement (ixaip.pin)
2. The results of placing and routing (ixaip.pli)
3. The design validation results (ixaip.vld)

In the validation results, there are 15 warnings reported related to fanouts over 10. These generally resulted from the use of Actel library macros which internally generate large fanouts, and therefore could not be corrected. These did not seem to cause any problems (except possibly to limit the maximum speed to 4MHz).
1. Automatic Pin Placement

; HEADER
; FILEID PLACEPIN ./ixaip/ixaip.pin 36d62c61
; CHECKSUM 36d62c61
; PROGRAM certify
; VERSION 1.14
; NODEID 0003ed5a
; DEPEND DDFFAMILY /user/als/data/a1000/family.ddf 6062d6e3
; DEPEND DDFDIE /user/als/data/a1000/a1020a/die1020a.ddf 362672b9
; DEPEND DDPACKAGE /user/als/data/a1000/a1020a/lcc68.ddf 61c2ce59
; DEPEND PlaceAFL ./ixaip/ixaip.afl 778da8e2
; DEPEND PlaceIPF ./ixaip/ixaip.ipf 95cc23f2
; VAR DDFDIE /user/als/data/a1000/a1020a/die1020a.ddf
; VAR DDFFAMILY /user/als/data/a1000/family.ddf
; VAR DDPACKAGE /user/als/data/a1000/a1020a/lcc68.ddf
; VAR PLACEAFL ./ixaip/ixaip.afl
; VAR DVDCRT ./ixaip/ixaip.crt
; VAR ADLIB /user/als/data/a1000/ad105.lib
; VAR PLACEIPF ./ixaip/ixaip.ipf
; VAR RESTRICTPROBEpins 1
; VAR DEFSYS /user/als/data/system.def
; VAR DEFSYS /user/mdd2257/mdd2257.def
; VAR DEFSYS /ixaip/ixaip.def
; VAR QUIETDEFOPENFAIL 1
; ENDHEADER
DEF ixaip.
NET 07;;
    PIN:8.
NET 11;;
    PIN:12.
NET 18;;
    PIN:16.
NET 01;;
    PIN:53.
NET OHO;;
    PIN:19.
NET EXTCLK;;
    PIN:22.
NET 04;;
    PIN:60.
NET 14;;
    PIN:3.
NET SWV;;
    PIN:50.
NET 15;;
    PIN:5.
NET 12;;
    PIN:6.
NET IHO;;
    PIN:7.
NET SWZ;;
    PIN:51.
NET SWC;;
    PIN:48.
NET 03;
   PIN: 52.
NET 02;
   PIN: 61.
NET 10;
   PIN: 13.
NET 08;
   PIN: 62.
NET IHI;
   PIN: 9.
NET I7;
   PIN: 10.
NET 00;
   PIN: 63.
NET 13;
   PIN: 18.
NET 05;
   PIN: 64.
NET SWS;
   PIN: 67.
NET OHI;
   PIN: 20.
NET IPRESET;
   PIN: 17.
NET 16;
   PIN: 68.
NET SWN;
   PIN: 47.
NET 06;
   PIN: 65.
END.
2. Placement Information

```
; HEADER
; FILEID PLACEINFO ./ixaip/ixaip.pli 8461c246
; CHECKSUM 8461c246
; PROGRAM certify
; VERSION 1.14
; NODEID 0003ed5a
; DEPEND DDFFAMILY /user/als/data/a1000/family.ddf 6062d6e3
; DEPEND DDFDIE /user/als/data/a1000/a1020a/diel0102a.ddf 362672b9
; DEPEND DDFPACKAGE /user/als/data/a1000/a1020a/lcc68.ddf 61c2ce59
; DEPEND PLACEAFL ./ixaip/ixaip.afl 778da8e2
; DEPEND PLACEPIN ./ixaip/ixaip.pin 36d62c61
; VAR DDFDIE /user/als/data/a1000/a1020a/diel0102a.ddf
; VAR DDFFAMILY /user/als/data/a1000/family.ddf
; VAR DDPACKAGE /user/als/data/a1000/a1020a/lcc68.ddf
; VAR PLACEAFL ./ixaip/ixaip.afl
; VAR DVDCRT ./ixaip/ixaip.crt
; VAR ADLIB /user/als/data/a1000/adl05.1ib
; VAR PLACEPIN ./ixaip/ixaip.pin
; VAR PLACEIPF ./ixaip/ixaip.ipf
; VAR PLACEABSOLUTECRITCOEFF <NOT-SET>
; VAR PLACEABSOLUTEFASTCRITCOEFF <NOT-SET>
; VAR PLACEVERTUNCRTCOEFF <NOT-SET>
; VAR PLACEHORUNCRTCOEFF <NOT-SET>
; VAR PLACEVERTCRITCOEFF <NOT-SET>
; VAR PLACEHORCRITCOEFF <NOT-SET>
; VAR PLACEHORFASTCRITCOEFF <NOT-SET>
; VAR PLACECLOCKBALANCESTRENGTH 0
; ENDHEADER

Average horizontal net length is 7.19751 columns.
A total of 39 nets need long horizontal tracks.
A total of 31 nets need long vertical tracks.
Automatic Placement completed successfully

The design has 481 routed (excluding GND, VCC, global CLKs) nets.
450 nets need no long vertical tracks,
av. horiz. length for these nets is 6.0178, av. fanout 2.6067.

31 nets need to use long vertical routing tracks.
31 of such tracks are of the standard type (LVT),
av. horiz. length for these nets is 24.3226, av. fanout 7.5161.

List of all nets using standard long vertical tracks (LVT) to route:

The net I$96/N$667 driven at location XY = (3, 9) uses an LVT.
LVT data: column = 20, Y-span = (8, 0).
Net data: fanout = 8, Y-spread of inputs = (9, 1).

The net I$96/N$64 driven at location XY = (12, 13) uses an LVT.
LVT data: column = 12, Y-span = (14, 0).
Net data: fanout = 1, Y-spread of inputs = (4, 4).
```
The net I$96/N$49 driven at location XY = (15,13) uses an LVT.
LVT data: column = 19, Y-span = (14, 0).
Net data: fanout = 1, Y-spread of inputs = ( 5, 5).

The net I$96/N$320 driven at location XY = (16, 2) uses an LVT.
LVT data: column = 7, Y-span = ( 7, 0).
Net data: fanout = 3, Y-spread of inputs = ( 7, 3)

The net I$96/N$169 driven at location XY = (13, 2) uses an LVT.
LVT data: column = 13, Y-span = ( 8, 0).
Net data: fanout = 4, Y-spread of inputs = ( 7, 1).

The net I$95/N$362 driven at location XY = (12, 5) uses an LVT.
LVT data: column = 31, Y-span = (14, 0).
Net data: fanout = 12, Y-spread of inputs = (13, 0)

The net I$95/N$361 driven at location XY = (18, 1) uses an LVT.
LVT data: column = 24, Y-span = (14, 0).
Net data: fanout = 12, Y-spread of inputs = (13, 1).

The net I$95/N$34 driven at location XY = (33, 1) uses an LVT.
LVT data: column = 29, Y-span = (14, 0).
Net data: fanout = 16, Y-spread of inputs = (12, 0).

The net I$95/N$30 driven at location XY = (37, 1) uses an LVT.
LVT data: column = 35, Y-span = (14, 0).
Net data: fanout = 16, Y-spread of inputs = (12, 0).

The net I$95/N$26 driven at location XY = (38, 1) uses an LVT.
LVT data: column = 40, Y-span = (14, 0).
Net data: fanout = 16, Y-spread of inputs = (13, 0).

The net I$95/N$22 driven at location XY = (31, 0) uses an LVT.
LVT data: column = 37, Y-span = (14, 0).
Net data: fanout = 16, Y-spread of inputs = (13, 0).

The net N$99 driven at location XY = (11,11) uses an LVT.
LVT data: column = 11, Y-span = (14, 0).
Net data: fanout = 3, Y-spread of inputs = (12, 1).

The net N$97 driven at location XY = (14,11) uses an LVT.
LVT data: column = 6, Y-span = (14, 0).
Net data: fanout = 3, Y-spread of inputs = (12, 1).

The net N$96 driven at location XY = (17,11) uses an LVT.
LVT data: column = 25, Y-span = (14, 0).
Net data: fanout = 3, Y-spread of inputs = (12, 1).

The net N$945 driven at location XY = (19, 3) uses an LVT.
LVT data: column = 33, Y-span = ( 8, 0).
Net data: fanout = 16, Y-spread of inputs = ( 8, 3).

The net N$944 driven at location XY = ( 4,10) uses an LVT.
LVT data: column = 14, Y-span = (14, 0).
Net data: fanout = 10, Y-spread of inputs = (13, 2).
The net N$88 driven at location XY = (28, 7) uses an LVT.
LVT data: column = 32, Y-span = (14, 0).
Net data: fanout = 2, Y-spread of inputs = (10, 1).

The net N$821 driven at location XY = (11,13) uses an LVT.
LVT data: column = 15, Y-span = (14, 0).
Net data: fanout = 11, Y-spread of inputs = (14, 2).

The net N$578 driven at location XY = ( 0, 4) uses an LVT.
LVT data: column = 2, Y-span = (14, 0).
Net data: fanout = 13, Y-spread of inputs = (10, 6).

The net N$336 driven at location XY = (20, 2) uses an LVT.
LVT data: column = 16, Y-span = (14, 0).
Net data: fanout = 1, Y-spread of inputs = ( 8, 9).

The net N$288 driven at location XY = (23,12) uses an LVT.
LVT data: column = 18, Y-span = (14, 0).
Net data: fanout = 2, Y-spread of inputs = ( 5, 5).

The net N$284 driven at location XY = (22, 9) uses an LVT.
LVT data: column = 22, Y-span = (14, 0).
Net data: fanout = 2, Y-spread of inputs = ( 6, 5).

The net N$249 driven at location XY = (20, 0) uses an LVT.
LVT data: column = 27, Y-span = (14, 0).
Net data: fanout = 8, Y-spread of inputs = (12, 4).

The net N$248 driven at location XY = ( 5, 5) uses an LVT.
LVT data: column = 28, Y-span = (14, 0).
Net data: fanout = 8, Y-spread of inputs = (10, 1).

The net N$233 driven at location XY = ( 4, 4) uses an LVT.
LVT data: column = 10, Y-span = (14, 6).
Net data: fanout = 3, Y-spread of inputs = (13, 5).

The net N$230 driven at location XY = ( 6,13) uses an LVT.
LVT data: column = 5, Y-span = (14, 0).
Net data: fanout = 8, Y-spread of inputs = (14, 2).

The net N$180 driven at location XY = ( 8, 6) uses an LVT.
LVT data: column = 8, Y-span = (14, 0).
Net data: fanout = 4, Y-spread of inputs = ( 9, 3).

The net N$178 driven at location XY = (21, 9) uses an LVT.
LVT data: column = 34, Y-span = (14, 0).
Net data: fanout = 6, Y-spread of inputs = (10, 4).

The net N$1043 driven at location XY = (14,13) uses an LVT.
LVT data: column = 21, Y-span = (14, 0).
Net data: fanout = 10, Y-spread of inputs = (14, 3).

The net N$102 driven at location XY = ( 0, 9) uses an LVT.
LVT data: column = 9, Y-span = (14, 0).
Net data: fanout = 10, Y-spread of inputs = (13, 1).

The net N$101$ driven at location XY = (6,11) uses an LVT.
LVT data: column = 3, Y-span = (14, 0).
Net data: fanout = 5, Y-spread of inputs = (13, 5).
3. Validation File (ixaip.vld)

; HEADER
; FILEID DVDCHK ./ixaip/ixaip.vld ac6f3b89
; CHECKSUM ac6f3b89
; PROGRAM certify
; VERSION 1.14
; NODEID 0003ed5a
; DEPEND DVDCRT ./ixaip/ixaip.crt 95cc23f2
; DEPEND DVDAFL ./ixaip/ixaip.afl 778da8e2
; DEPEND DVDPF ./ixaip/ixaip.ipf 95cc23f2
; VAR DVDCRT ./ixaip/ixaip.crt
; VAR DVDAFL ./ixaip/ixaip.afl
; VAR DVDPF ./ixaip/ixaip.ipf
; ENDHEADER
Design variables:
  DESIGN = ixap
  DIE = 1020a
  PACKAGE = lcc68
  ADLIB = /user/als/data/a1000/ad105.lib

Loaded design netlist.
Loaded pin locations.
Loaded net criticalities.

Beginning Pass 1...
Checking definition ADLIB:MX4
Checking definition ADLIB:AND2
Checking definition ADLIB:OR2
Checking definition ADLIB:AND3B
Checking definition ADLIB:XOR
Checking definition ADLIB:AND3A
Checking definition ADLIB:GND
Checking definition ADLIB:AND2A
Checking definition ADLIB:DFE1B_0
Checking definition ADLIB:DFE1B_1
Checking definition ADLIB:DFE1B
Checking definition ADLIB:TA377
Checking definition ADLIB:AND3
Checking definition DS
Checking definition ADLIB:OUTBUF
Checking definition ADLIB:INBUF
Checking definition ADLIB:MX2
Checking definition ADLIB:BUF
Checking definition ADLIB:AO1
Checking definition ALUBIT
Checking definition ADLIB:OR3
Checking definition ADLIB:XNOR
Checking definition ADLIB:OR4
Checking definition ADLIB:NOR2
Checking definition ADLIB:XA1
Checking definition ALU
Checking definition ADLIB:AO1A
Checking definition ADLIB:AND4A_0
Checking definition ADLIB:AND4A_1
Checking definition ADLIB:AND4A...
Checking definition ADLIB:DLC_0...
Checking definition ADLIB:DLC...
Checking definition ADLIB:DLC8A...
Checking definition ADLIB:DFM3_0...
Checking definition ADLIB:DFM3_1...
Checking definition ADLIB:DFM3...
Checking definition ADLIB:DFC1A_0...
Checking definition ADLIB:DFC1A_1...
Checking definition ADLIB:DFC1A...
Checking definition CLKPHI...
Checking definition ADLIB:0R2A...
Checking definition ADLIB:OA1...
Checking definition ADLIB:KO1...
Checking definition ADLIB:AND4C...
Checking definition ADLIB:AND4B...
Checking definition ADLIB:INV...
Checking definition ADLIB:A02...
Checking definition ADLIB:AND4D_0...
Checking definition ADLIB:AND4D_1...
Checking definition ADLIB:AND4D...
Checking definition ADLIB:MX8...
Checking definition ADLIB:A03...
Checking definition CU...
Checking definition ADLIB:DFC1_0...
Checking definition ADLIB:DFC1_1...
Checking definition ADLIB:DFC1...
Checking definition REGBLK...
Checking definition IOU...
Checking definition ixai...
Checking global and external connections...
Checking physical consistency
Calling the Combiner...
Beginning Pass 2...
Checking definition ADLIB:AND2, used 67 times...
Checking definition ADLIB:DFE1B_0, used 48 times...
Checking definition ADLIB:DFE1B_1, used 48 times...
Checking definition ADLIB:DFE1B, used 48 times...
Checking definition ADLIB:A01, used 20 times...
Checking definition ADLIB:BUF, used 13 times...
Checking definition ADLIB:MX2, used 37 times...
Checking definition ADLIB:AND3A, used 17 times...
Checking definition ADLIB:OR3, used 14 times...
Checking definition ADLIB:A01A, used 3 times...
Checking definition ADLIB:DLC_0, used 24 times..
Checking definition ADLIB:DLC, used 24 times..
Checking definition ADLIB:XOR, used 37 times...
Checking definition ADLIB:INV, used 5 times...
Checking definition ADLIB:OUTBUF, used 16 times.
Checking definition ADLIB:OR2, used 19 times...
Checking definition ADLIB:INBUF, used 13 times...
Checking definition ADLIB:DFM3_0, used 3 times...
Checking definition ADLIB:DFM3_1, used 3 times...
Checking definition ADLIB:DFM3, used 3 times...
Checking definition ADLIB:DFM3, used 3 times...
Checking definition ADLIB:XNOR, used 2 times...
Checking definition ADLIB:MX4, used 18 times...
Checking definition ADLIB:AND2A, used 20 times...
Checking definition ADLIB:XA1...
Checking definition ADLIB:DFC1A_0, used 4 times...
Checking definition ADLIB:DFC1A_1, used 4 times...
Checking definition ADLIB:AND3, used 11 times...
Checking definition ADLIB:AND3B, used 6 times...
Checking definition ADLIB:OR4, used 9 times...
Checking definition ADLIB:DFC1_0, used 10 times...
Checking definition ADLIB:DFC1_1, used 10 times...
Checking definition ADLIB:DFC1, used 10 times..
Checking definition ADLIB:AND4, used 11 times..
Checking definition ADLIB:AND4B, used 3 times..
Checking definition ADLIB:0R2A...
Checking definition ADLIB:X01...
Checking definition ADLIB:NOR2...
Checking definition ADLIB:OA1...
Checking definition ADLIB:AND4A_0...
Checking definition ADLIB:AND4A_1...
Checking definition ADLIB:AND4A...
Checking definition ADLIB:AND4D_0...
Checking definition ADLIB:AND4D_1...
Checking definition ADLIB:AND4D...
Checking definition ADLIB:AO3...
Checking definition ADLIB:AND4C, used 2 times...
Checking definition ADLIB:AO2...
Checking definition ixaip...
Checking global and external connections...
VLD3202: WARNING: Fanout of 12 exceeds recommended limit of 10.
    Net N$228
VLD3202: WARNING: Fanout of 16 exceeds recommended limit of 10.
    Net N$239
VLD3202: WARNING: Fanout of 13 exceeds recommended limit of 10.
    Net N$578
VLD3202: WARNING: Fanout of 12 exceeds recommended limit of 10.
    Net N$804
    Net N$81
    Net N$821
    Net N$944
VLD3202: WARNING: Fanout of 16 exceeds recommended limit of 10.
    Net N$945
VLD3202: WARNING: Fanout of 16 exceeds recommended limit of 10.
    Net I9S5/N$22
VLD3202: WARNING: Fanout of 16 exceeds recommended limit of 10.
    Net I9S5/N$26
VLD3202: WARNING: Fanout of 16 exceeds recommended limit of 10.
    Net I9S5/N$30
VLD3202: WARNING: Fanout of 16 exceeds recommended limit of 10.
    Net I9S5/N$34
VLD3202: WARNING: Fanout of 12 exceeds recommended limit of 10.
    Net I9S5/N$361
VLD3202: WARNING: Fanout of 12 exceeds recommended limit of 10.
Net I$95/N$362
Net I$96/N$701
Checking physical consistency...
VLD5300: Design uses:
- 468 logic modules
- 29 io modules

1410 module inputs driven by 481 signal nets with 2.93 average fanout.

0 errors and 15 warnings.

VLD2600: This design contains 15 nets with fanouts greater than the recommended limit of 10. You are strongly advised to keep net fanouts below 10.
VLD2601: The probability of successfully routing this design is 91%.  
Appendix I - MC68008-based Program Controller

This appendix contains an assembler listing of the simple program controller written for the MC68008 board which was used with the FPGA-based processor. The program uses the parallel output chip on the board to implement the handshaking and data transfer to the processor. It reads data back from the chip after any WRITE instruction is sent. The program to be sent is stored as a memory image in the 68000 code; this can be modified by overlaying other programs. The number -1 (FFFF hex) is used to signal the end of the program.
*** IXA Test Bench Program Controller ***
*** Matthew Dimmore, 1993 ***
*** Version 1.1.0 ***

*** Implements a basic program controller on a 68008 ***
*** board using IXA handshaking protocol ***

*** Interface runs in Bit I/O Mode, with Port A and the ***
*** LSB of Port B set for output, and the remainder of ***
*** Port B set for input (for use with future applications). ***
*** Port C is used to input data. Data is stored starting at 1500. ***

*** Connection Diagram

*** 68008 Board       IXAIP108
*** PORT A [0:7] -> PORT 0[0:7] IN
*** PORT B [0]   -> PORT 0[8] IN
*** PORT C [0:7]   -> PORT 0[8] OUT

***     H1 <- IHo
***     H2 -> IHi
***     H3 <- OHo
***     H4 -> OHi

*** EQUATES

=000FF000  PGCR   EQU  $FF000
=000FF00C  PACR   EQU  $FF00C
000001000 ORG $1000
<table>
<thead>
<tr>
<th>E</th>
<th>Seq</th>
<th>Address</th>
<th>Code</th>
<th>Source Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>00001000</td>
<td>13FC 00 3F</td>
<td>INIT</td>
<td>MOVE.B #$3F,PGCR ;MODE 0,H1234 EMA, ASSERTED HIGH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00FF000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>00001008</td>
<td>13FC 00 A0</td>
<td></td>
<td>MOVE.B #$A0,PACR ;SUBMODE 1X, H2 NEGATED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00FF00C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>00001010</td>
<td>13FC 00 A0</td>
<td></td>
<td>MOVE.B #$A0,PBCR ;SUBMODE 1X, H2 NEGATED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00FF00E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>00001018</td>
<td>13FC 00 FF</td>
<td></td>
<td>MOVE.B #$FF,PADDR ;ALL PINS OUTPUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00FF004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>00001020</td>
<td>13FC 00 03</td>
<td></td>
<td>MOVE.B #$03,PBADDR ;2 LSBs OUT, OTHERS IN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00FF006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>00001028</td>
<td>13FC 00 00</td>
<td></td>
<td>MOVE.B #$00,PCADDR ;USE C FOR INPUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00FF008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>00001030</td>
<td>207C 0001200</td>
<td></td>
<td>MOVEA.L #CODEIN,A0 ;ADDRESS OF PROGRAM</td>
</tr>
<tr>
<td>62</td>
<td>00001036</td>
<td>227C 00001500</td>
<td></td>
<td>MOVEA.L #CODEOUT,A1 ;ADDRESS OF RESULTS</td>
</tr>
<tr>
<td>63</td>
<td>0000103C</td>
<td>383C 0000</td>
<td></td>
<td>MOVE.W #0,D4 ;INIT COUNT</td>
</tr>
<tr>
<td>64</td>
<td>00001040</td>
<td>3018</td>
<td></td>
<td>MOVE.W (A0)+,D0 ;SKIP COUNT WORD</td>
</tr>
<tr>
<td>65</td>
<td>00001042</td>
<td>B879 00001200</td>
<td>START</td>
<td>CMP.W CODEIN,D4 ;TEST FOR END</td>
</tr>
<tr>
<td>66</td>
<td>00001048</td>
<td>6700 04B8</td>
<td></td>
<td>BEQ ALLDONE ;ALL DONE</td>
</tr>
<tr>
<td>67</td>
<td>0000104C</td>
<td>3018</td>
<td></td>
<td>MOVE.W (A0)+,D0 ;GET AN INSTRUCTION</td>
</tr>
<tr>
<td>68</td>
<td>0000104E</td>
<td>3600</td>
<td></td>
<td>MOVE.W D0,D3 ;NEED FOR TEST LATER</td>
</tr>
<tr>
<td>69</td>
<td>00001050</td>
<td>5244</td>
<td></td>
<td>ADD.W #1,D4 ;INC COUNT</td>
</tr>
<tr>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>00001052</td>
<td>13C0 000FF010</td>
<td>PAROUT</td>
<td>MOVE.B D0,PADR ;MOVE MSB TO PORT A</td>
</tr>
<tr>
<td>72</td>
<td>00001058</td>
<td>E048</td>
<td></td>
<td>LSR.W #8,D0 ;SHIFT MSB TO LSB</td>
</tr>
<tr>
<td>73</td>
<td>0000105A</td>
<td>13C0 000FF012</td>
<td></td>
<td>MOVE.B D0,PBDR ;MOVE TO PORT B</td>
</tr>
<tr>
<td>74</td>
<td>00001060</td>
<td>13FC 00 A8</td>
<td></td>
<td>MOVE.B #$A8,PACR ;ASSERT H2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00FF00C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
75 00001068 1239 00FF01A PAROUTWAIT MOVE.B PSR,D1 ;CHECK PARALLEL HANDSHAKE
76 0000106E 0801 0004 BTST.B #4,D1 ;INSTANT LEVEL OF H1
77 00001072 67F4 BEQ PAROUTWAIT ;WAIT UNTIL PROCESSOR ACKS
78 00001074 13FC 00 A0 MOVE.B #$A0,PACR ;DROP H2
00FF00C
79 0000107C 1239 00FF01A MOREWAIT MOVE.B PSR,D1 ;NOW WAIT FOR IT TO DROP
80 00001082 0801 0004 BTST.B #4,D1 ;LOOK AT H1 INSTANT
81 00001086 66F4 BNE MOREWAIT ;NOT YET
82 00001088 60B8 BRA START
83 0000108A 0C43 0184 CMP.W #$184,D3 ;IS INSTRUCTION A WRIT?
84 0000108E 6700 000C BEQ TESTPAR ;YES - GOTO INPUT
85 00001092 0C43 0188 CMP.W #$188,D3 ;OR WRITC?
86 00001096 6700 0004 BBQ TESTPAR ;YES - GOTO INPUT
87 0000109A 60A6 BRA START ;GOTO NEXT OUTPUT
88
89 0000109C 1239 00FF01A TESTPAR MOVE.B PSR,D1 ;CHECK TO SEE IF H3 IS SET (HANDSHAKE IN)
90 000010A2 0801 0006 BTST.B #6,D1 ;IS H3 ASSERTED?
91 000010A6 67F4 BBQ TESTPAR ;NO - WAIT FOR IT
92 000010A8 1439 00FF018 MOVE.B PCDR,D2 ;GET THE DATA TO D2
93 000010AE 13FC 00 A8 MOVE.B #$A8,PBCR ;ACK DATA WITH H4 HANDSHAKE
00FF00E
94 000010B6 12C2 MOVE.B D2,(A1)+ ;STORE INPUT IN MEMORY
95 000010BB 1239 00FF01A PARINWAIT MOVE.B PSR,D1 ;CHECK TO SEE IF PROC HAS DROPPED HS
96 000010BE 0801 0006 BTST.B #6,D1 ;H3 CLEARED?
97 000010C2 66F4 BNE PARINWAIT ;NO - POLL IT AGAIN
98 000010C4 13FC 00 A0 MOVE.B #$A0,PACR ;DROP H4 HANDSHAKE
00FF00E
<table>
<thead>
<tr>
<th>E</th>
<th>Seq</th>
<th>Address</th>
<th>Code</th>
<th>Source Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>000010CC</td>
<td>6000 FF74</td>
<td></td>
<td>BRA START ;SEND NEXT WORD</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>=00001200</td>
<td></td>
<td>ORG $1200</td>
</tr>
<tr>
<td>102</td>
<td>00001200</td>
<td>0006</td>
<td>CODEIN</td>
<td>DC.W $6</td>
</tr>
<tr>
<td>103</td>
<td>00001202</td>
<td>0001</td>
<td>DC.W</td>
<td>$01</td>
</tr>
<tr>
<td>104</td>
<td>00001204</td>
<td>0188</td>
<td>DC.W</td>
<td>$188</td>
</tr>
<tr>
<td>105</td>
<td>00001206</td>
<td>0002</td>
<td>DC.W</td>
<td>$02</td>
</tr>
<tr>
<td>106</td>
<td>00001208</td>
<td>0188</td>
<td>DC.W</td>
<td>$188</td>
</tr>
<tr>
<td>107</td>
<td>0000120A</td>
<td>0150</td>
<td>DC.W</td>
<td>$150</td>
</tr>
<tr>
<td>108</td>
<td>0000120C</td>
<td>0188</td>
<td>DC.W</td>
<td>$188</td>
</tr>
<tr>
<td>109</td>
<td>0000120E</td>
<td>1111</td>
<td>DC.W</td>
<td>$1111</td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td>=00001500</td>
<td></td>
<td>ORG $1500</td>
</tr>
<tr>
<td>112</td>
<td>00001500</td>
<td>0000</td>
<td>CODEOUT</td>
<td>DC.W $00</td>
</tr>
<tr>
<td>113</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>00001502</td>
<td>4E72 000A</td>
<td>ALLDONE</td>
<td>STOP #10</td>
</tr>
<tr>
<td>115</td>
<td></td>
<td></td>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>
* * Cross Reference * *

References (# = Definition, $ = Write, <Blank> = Read)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALDDONE</td>
<td>00001502</td>
<td>2-066 3-114#</td>
</tr>
<tr>
<td>CODEIN</td>
<td>00001200</td>
<td>2-061 2-065 3-102#</td>
</tr>
<tr>
<td>CODEOUT</td>
<td>00001500</td>
<td>2-062 3-112#</td>
</tr>
<tr>
<td>INIT</td>
<td>00001000</td>
<td>1-055#</td>
</tr>
<tr>
<td>MOREWAIT</td>
<td>0000107C</td>
<td>2-079# 2-081</td>
</tr>
<tr>
<td>PACR</td>
<td>=00FF00C</td>
<td>1-028# 2-056$ 2-074$ 2-078$</td>
</tr>
<tr>
<td>PADDR</td>
<td>=00FF004</td>
<td>1-030# 2-058$</td>
</tr>
<tr>
<td>PADR</td>
<td>=00FF010</td>
<td>1-033# 2-071$</td>
</tr>
<tr>
<td>PARINWAIT</td>
<td>000010B8</td>
<td>2-095# 2-097</td>
</tr>
<tr>
<td>PAROUT</td>
<td>00001052</td>
<td>2-071#</td>
</tr>
<tr>
<td>PAROUTW</td>
<td>00001068</td>
<td>2-075# 2-077</td>
</tr>
<tr>
<td>PBCR</td>
<td>=00FF00E</td>
<td>1-029# 2-057$ 2-093$ 2-098$</td>
</tr>
<tr>
<td>PBDDR</td>
<td>=00FF006</td>
<td>1-031# 2-059$</td>
</tr>
<tr>
<td>PBDR</td>
<td>=00FF012</td>
<td>1-034# 2-073$</td>
</tr>
<tr>
<td>PCDDDR</td>
<td>=00FF008</td>
<td>1-032# 2-060$</td>
</tr>
<tr>
<td>PCDR</td>
<td>=00FF018</td>
<td>1-035# 2-092</td>
</tr>
<tr>
<td>PCGR</td>
<td>=00FF000</td>
<td>1-027# 1-055$</td>
</tr>
<tr>
<td>PSR</td>
<td>=00FF01A</td>
<td>1-036# 2-075 2-079 2-089 2-095</td>
</tr>
<tr>
<td>RHRA</td>
<td>=00FF806</td>
<td>1-039#</td>
</tr>
<tr>
<td>RHRB</td>
<td>=00FF816</td>
<td>1-042#</td>
</tr>
<tr>
<td>SRA</td>
<td>=00FF802</td>
<td>1-038#</td>
</tr>
<tr>
<td>SRB</td>
<td>=00FF814</td>
<td>1-041#</td>
</tr>
<tr>
<td>START</td>
<td>0000104:2</td>
<td>2-065# 2-082 2-119 3-099</td>
</tr>
</tbody>
</table>
Summary Statistics Produced By BSO/ASM68000 Version 1.8

Assembler Invocation: ASM/ABSOLUTE/DEBUG/LISTING=SPIF.LST SPIF.ASM

Module name: .MAIN.

Elapsed Time: 00:00:01.96
CPU Usage: 00:00:00.34
Buffered IO: 24
Direct IO: 27
Page Faults: 418

No Errors Encountered