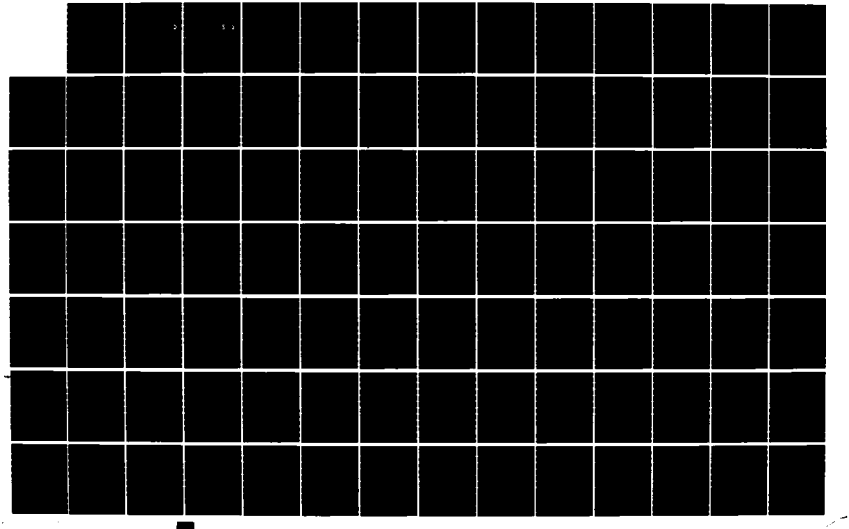
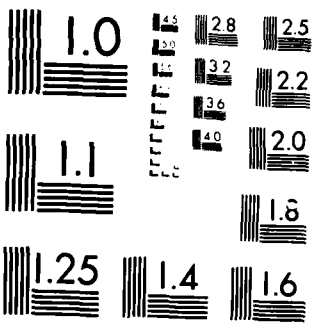
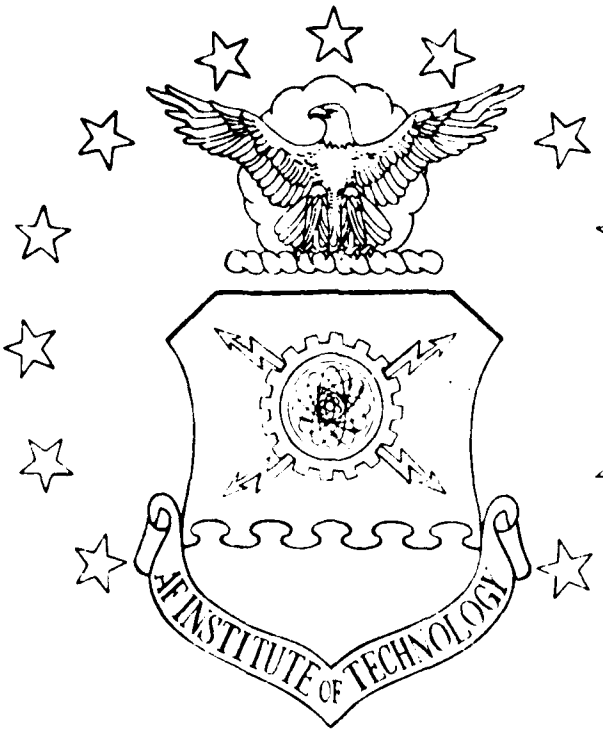
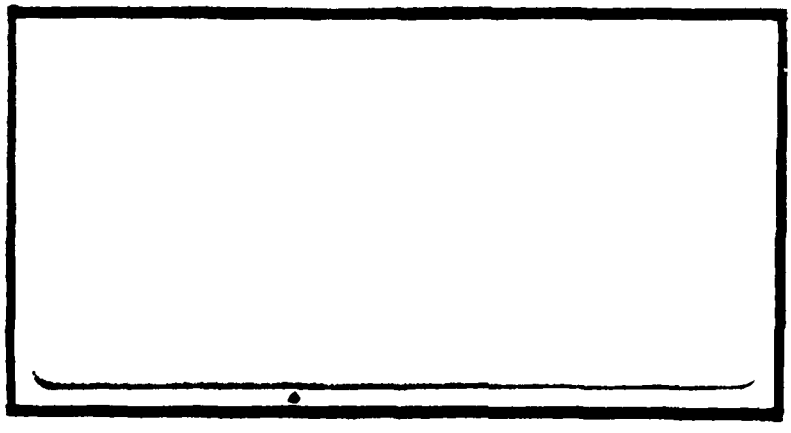
AD-A164 203 THE SIMULATION AND ANALYSIS OF A RTL MODEL OF THE MOTOROLA MC68000 MICROP..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. C A BAXLEY 1/2

UNCLASSIFIED DEC 84 AFIT/GCS/ENG/84D-2-VOL-1 F/G 9/2 NL









DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

(1)

AFIT/GCS/ENG/84D-2 - Vol. 1

THE SIMULATION AND ANALYSIS OF A RTL MODEL OF THE MOTOROLA MC68000 MICROPROCESSOR WITH N.MPC. Volume 1

(THESIS
(1 of 3))

Charles A. Baxley Jr.
Captain, USAF

AFIT/GCS/ENG/84D-2 - Vol. 1

AFIT/GCS/ENG/84D-2 - Vol. 1

THE SIMULATION AND ANALYSIS OF A RTL MODEL OF THE MOTOROLA MC68000 MICROPROCESSOR WITH N.MPC

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Charles A. Baxley Jr., B.S.
Captain, USAF

December 1984

Approved for public release; distribution unlimited

Preface

With the increasing complexity of today's microprocessors, gate level analysis and testing is no longer practical. In addition, since the Air Force is now using off-the-shelf microprocessors, manufacturing details are not always available. In an earlier thesis, Captain James Hamby and Lt Galen Guillory presented a method for analyzing and modeling these new microprocessors at the functional level without using manufacturing information. In this follow-on thesis, I document the simulation of those microprocessor models developed by Hamby and Guillory and present an evaluation of their modeling approach based on the results. I selected this topic because it provided an opportunity to apply much of the course material taken while at AFIT, it introduced me to new areas in the field of computer architecture, and it enabled me to examine a valid problem facing today's Air Force.

I would like to extend my sincere appreciation to my Thesis Advisor, Major Fred Zapka, who provided me with first-rate guidance whenever the direction of my work became clouded with technical and administrative problems. Without his expert instruction and consistent enthusiasm, I would certainly have fallen short of the major objectives of this work and the quality of this report would have diminished substantially. I would also like to thank my Thesis Reader, Lt Col Hal Carter, for his support; particularly for

acquiring the simulation package that made this work possible and assisting in its installation. I also thank Major Ken Melendez of the Foreign Technology Division (FTD) for his support as the sponsor of this project and hope that my work has been beneficial to FTD. Finally, I would like to thank my wife, Lois, and my son, Allen, for their endless patience, understanding, and support during those many hours we were separated by this work.

Contents

(Volume I)

List of Figures

Abstract

In a prior thesis project, a functional level model of portions of the Motorola MC68000 microprocessor was developed using signal analysis supported by limited technical data. Representative parts of the instruction set and exception processing structure were modeled with the Computer Design Language (CDL). In this follow-on effort, those CDL models are transformed into equivalent models using ISP', an enhanced version of the Instruction Set Processor (ISP) hardware design language. This language transformation enabled the models to be simulated using N.mPc, a VAX 11/780-hosted software package developed specifically to support the design of digital systems. To evaluate the correctness of the models, the simulation results are analyzed against signal data gathered with the aid of a logic analyzer during the actual operation of the MC68000 when processing the modeled instructions. The accuracy and completeness of the models suggest that this functional approach to microprocessor modeling is a valid one.

THE SIMULATION AND ANALYSIS OF A RTL MODEL OF THE MOTOROLA MC68000 WITH N.MPC

I. Introduction

Background

The number of off-the-shelf commercial microprocessors used in Air Force weapons systems is increasing at a dramatic rate. In contrast to specially-developed microprocessors designed for specific applications, off-the-shelf devices are readily available, less costly, and usually shorten system development time. However, they have one significant disadvantage - detailed technical data important to systems development is often unavailable. Technical data of interest include the schematic and logic diagrams, microcode descriptions, and production masks that are normally provided with specially-developed devices.

Because functional models of microprocessors are invaluable to their successful integration into modern weapons sytems, the Air Force has a strong interest in uncovering a method to develop functional microprocessor models without the benefit of the detailed technical data - the situation common to off-the-shelf device utilization.

In an effort to assist the Air Force in obtaining a solution to this important problem, Captain James R. Hamby and 1Lt Galen J. Guillory directed their thesis research to developing a functional model of the Motorola MC68000

microprocessor within the "data-poor" environment just described. Documented in their joint thesis entitled "Architectural Analysis and Modeling of A Motorola MC68000 Microprocessor," a functional model of the MC68000 was developed without the aid of any manufacturer's schematics or other technical data not readily available to a retail purchaser of the system (8:I-6). By successfully modeling the MC68000, Captain Hamby and 1Lt Guillory hoped to demonstrate that not only could a model be developed within the information constraints, but that such a model could also be constructed in an efficient and structured manner, within a reasonable period of time, and without encountering overwhelming technical difficulties.

A functional level model describes the register transfer operations within a device. It enables sytems developers to observe the internal timing and control of logic and register transfer operations long before the microprocessor is included in production systems. Because the detailed gate circuitry is not modeled, functional models are relatively easy to implement, understand, and maintain.

There were two basic approaches that Hamby and Guillory could have taken to analyze and develop a model of the MC68000. One commonly used method requires that the microprocessor be disassembled. During states of disassembly, electron microscope photographs are taken so that the actual circuitry of the chip can be reconstructed

from the photographs. A model can easily be constructed from the resulting logic diagrams and microcode. Although this method produces a very accurate model, it is a very time-consuming and complex process (8:I-8).

An untried, alternative approach required that the timing and voltage levels of the MC68000's pin signals be analyzed while the processor was in operation. The functions performed by the processor could then be modeled using a computer hardware design language. This approach promised faster and more efficient model development, but the accuracy of the resulting model was uncertain. Captain Hamby and 1Lt Guillory used this second, unchartered modeling approach to build their functional model.

Now that a functional model of the MC68000 has been developed within the imposed constraints, some basic questions naturally arise. Is this "blackbox" approach to microprocessor modeling a sound one? Can an accurate and complete functional model of a microprocessor that expresses the timing, control, parallelism, microprogramming, and other internal operations of today's complex microprocessors be developed by examining input and output signals only? If deficiencies are inherent in this modeling approach, can they be corrected, neglected, or compensated for? Or, are they numerous and significant enough to eliminate this approach as a viable modeling technique in favor of the first, more complex approach? Answers to these questions are essential before this approach can gain acceptance as

the solution to the modeling dilemma confronting the Air Force and other DoD agencies.

Problem

This research objective was to evaluate the approach to microprocessor modeling chosen by Captain Hamby and 1Lt Guillory by simulating their model and then analyzing the simulation results against the data they observed during the operation of an actual processor under equivalent conditions. The adequacy of the model will mirror the viability of their approach.

Scope

Due to time constraints, Captain Hamby and 1Lt Guillory did not model the entire microprocessor. Portions of the MC68000 modeled include the read and write bus cycles, representative instruction types, and exception processing sequences (8:II-1). The current model was not extended or optimized, nor were other microprocessors modeled so that broader inferences regarding the effectiveness of this approach to microprocessor modeling could be made. All effort was focused on examining and drawing conclusions from the existing partial model. Because this research yielded positive results, additional research aimed at completing the existing model as well as modeling other architectures will more conclusively demonstrate the practicality and applicability of this modeling approach.

Assumptions

N.mPc (network of microprocessors) is a register transfer level (RTL) simulation system that has been successfully used by government and industrial engineering shops in designing VLSI and multiple microprocessor systems (5:76). The validity of this research was heavily dependent upon a simulation package that is efficient, reliable, and most important, comprehensive and accurate. Because N.mPc was the only RTL simulation package available for this research, and its performance had not been personally observed, texts, periodicals, and the developer's documentation were used to vouch for its worthiness. Subsequently, an up-front assumption was that N.mPc would perform well, and it did.

Summary of Current Knowledge

Most (if not all) microprocessors have been modeled with computer hardware design languages, and many have been simulated with N.mPc. As a result, MC68000 simulations are available from which comparisons can be made to help determine the accuracy of the model constructed by Captain Hamby and 1Lt Guillory. However, the significance of this research is not centered around the simulation of their model, but rather their approach to model development. Up to now, attempts to develop functional microprocessor models using their "blackbox" approach have been negligible. If their approach can be validated, then this technique will become a great boon to governmental agencies and commercial

businesses employing microprocessor technology.

Approach

The approach to evaluating their modeling technique was basically sequential in nature. The simulation, and the subsequent evaluation and documentation of its results followed periods during which the Motorola MC68000 microprocessor, the N.mPc simulation package, both the CDL and ISP' computer hardware design languages, and the MC68000 Educational Computer Board (ECB) were learned. The solution steps to this research problem were:

1) Before the N.mPc software could be used to simulate the "Hamby and Guillory" model, it was first brought "on-line". The package had been delivered to AFIT via tape, but was not yet operational. Using the system documentation provided (i.e., installation and user's manuals), the system was successfully installed on our SSC VAX 11/780 for this thesis effort.

2) Captain Hamby and 1Lt Guillory selected the Computer Design Language (CDL) to describe their model because it was relatively simple to use and understand, and they were familiar with its structure. However, to simulate a microprocessor using N.mPc, it had to be described with a variation of the Instruction Set Processor (ISP) hardware design language. Therefore, both the ISP' and CDL hardware design languages had to be mastered before the necessary conversion could be accomplished.

3) Once an ISP' model was constructed, the operational

N.mPc simulator was used to exercise the model to generate an operational scenerio of the MC68000 as it executed the same instruction sequences selected by Hamby and Guillory. Chapter VI of their thesis outlines these instructions.

4) The simulation results were then compared with the documented logic analyzer signals that described the ECB's MC68000 when executing equivalent instruction sequences. A functionally correct model would accurately project all event occurrences on the system bus. The simulated results should have coincided with that of the physical hardware at each clock cycle (7:459). Any differences required that the model be carefully examined for errors. If the model's microinstruction sequences were not at fault, then the microprocessor's actual operation was again monitored to determine if the logic analyzer data was incorrect. The 68000's ability to prefetch instructions and generate vectors were expected to be likely causes of model errors. Since the prefetch occurs in parallel with instruction execution, it was difficult for Hamby and Guillory to simultaneously monitor both events. The actual generation of vectors was difficult to analyze because this event is totally internally-accomplished (8:V-1,V-2). The time required to complete this portion of the research depended upon the adequacy of the model. Each difference triggered its own "trouble-shooting" session. Once all of the differences had been reconciled, the corrected model was simulated to produce a valid operational scenerio. The

final product of this step was a model that accurately reflected the MC68000's actual operation, a detailed log of any deficiencies detected, the source of these deficiencies, the corrections made, and an assessment of each deficiency's overall impact on the model.

5) Finally, the overall effectiveness of the model was assessed and documented based on the types and numbers of errors encountered, especially those that had a significant bearing on the feasibility of the modeling approach under study. Although this thesis played the "devil's advocate" and concentrated fully on reporting obstacles that may hinder or prohibit this approach from being used, the positive attributes of this model and approach are also fully documented.

Overview of Presentation

Chapter II next outlines the requirements of this thesis in detail. To familiarize the reader with both the Motorola MC68000 and the N.mPc simulation package, Chapters III and IV provide brief descriptions of each. Chapter V details the CDL-to-ISP' model transformation process. The N.mPc simulation of the MC68000 model and the analysis of its results are presented in Chapter VI. Finally, Chapter VII contains several conclusions and recommendations and is followed by a short chapter that documents the time spent on each phase of this thesis project.

II. Requirements

Introduction

This chapter elaborates further on the research approach by outlining the requirements of this project in detail. As mentioned, this effort required that the following five objectives be achieved;

1) install the N.mPc simulation package;

2) transform the existing CDL models into equivalent, simulatable ISP' models;

3) develop a simulation strategy that enables the observation of those same signals as monitored and documented by Hamby and Guillory;

4) simulate the models and then analyze and report the results, and finally;

5) prepare a detailed time log of all work associated with this effort.

N.mPc Installation

Release II of N.mPc was on-loaded onto AFIT's SSC VAX 11/780 from its delivered magnetic tape medium. After being on-loaded, the necessary directory structuring and file relocation was accomplished before the system could be brought on-line and operationally certified. Once operational, all unneccessary software such as documentation, superfluous ISP' library models, and test programs were removed to minimize N.mPc's demands on the VAX's limited storage space. N.mPc's system protection

mechanisms were also then reconfigured to enable access from this and other interested user's login directories.

Model Transformation

Because the scope of Hamby and Guillory's research did not include consolidating or generalizing their numerous instruction descriptions, they in effect developed a single model of the MC68000 hardware elements that was accompanied by multiple, independent models of the following instructions and exceptions: MOVE, JMP, ADD, BEQ, BTST, Illegal Instruction, and Address Error. These models could be individually appended to the lone hardware description to form individually simulatable instructions and exception conditions.

To simplify the model transformation process, this same level of development was preserved. A single ISP' description of the hardware elements was developed to support multiple instruction or exception processing models. This "multiple model" approach simplified model development, made the size of the model manageable, aided in the debugging process, and helped approximate a one-to-one correspondence between CDL and ISP' statements so that model equivalency could more readily be established.

In order to execute the assembly language test routines on the MC68000 ISP' model, the 68000's instruction set had to be described via N.mPc's assembler component, Metamicro. Metamicro allows its users to define the structure and semantics of the target processor's instruction set so that

source programs designed to run on that processor can be written and assembled. A second N.mPc component, the Linking/Loader, then allows the user to load programs assembled by Metamicro into the microprocessor's simulated memory for execution. Rather than develop auxillary Metamicro and Linking/Loader routines to construct a workable simulation, these routines were extracted from N.mPc's microprocessor library. With minor modifications, they supported the assembling and loading of the test routines into the MC68000 model.

Simulation Strategy

To develop their models, Hamby and Guillory used a logic analyzer to monitor numerous MC68000 signals that provided information pertinent to instruction processing. These signals varied slightly, depending on which instruction or exception sequence was modeled, but at most included the following 20 signals:

1 - FC0
2 - FC1
3 - FC2
4 - DTACK'
5 - R/W
6 - LDS'
7 - UDS'
8 - AS'
9 - D0
10 - D1
11 - D2
12 - D3
13 - D4
14 - D5
15 - D6
16 - D7
17 - A1

18 - A2
19 - A3
20 - A4

These signals were examined on both the positive and negative transitions of each clock signal to emulate a two-phase clock and therefore increase the operational resolution of their model's timing. To accurately compare and contrast the simulation results with the observed data, it was essential that a simulation strategy be developed for each instruction or exception sequence that enabled the observation of these same signals at the positive and negative transitions of each simulated clock cycle.

Simulation Analysis

The methodology of the simulation analysis phase has been carefully documented in steps 4 and 5 of the research approach (pages I-7 and I-8) and will not be heavily expanded here. However, it should be noted that existing models of the MC68000 that are resident in N.mPc's microprocessor library were used in conjunction with the documented logic analyzer results and ECB operation to assist in the identification of any model discrepancies and their corresponding remedies.

Detailed Time Log

As done during Hamby's and Guillory's thesis effort, a detailed time log was maintained and then analyzed at the conclusion of this project. It too includes information on the time spent during each phase of this project such as

researching, writing, modeling, analyzing, and so on.

III. Introduction to the Motorola MC68000 Microprocessor

Introduction

A basic understanding of the M68000's architectural features and its capabilities will prepare the reader for the material of later chapters that describe the development and simulation of the M68000 model. This chapter introduces the M68000 with descriptions of its signals, register organization, instruction set, system architecture, data types, and addressing modes.

M68000 Introduction

The MC68000 is a 16-bit microprogrammed microprocessor with a 32-bit internal architecture. First introduced by Motorola in late 1979, this VLSI microprocessor combines state-of-the art technogy (HMOS) and advanced design techniques to achieve very fast processing speeds and high circuit densities (11:1). The chip contains approximately 68000 transistors (hence it's name) and is available in several different operating frequency versions. The 4, 6, 8, and 10-MHz versions have respective clock cycles of 250, 167, 125, and 100 ns.

The 68000's microprogrammed architecture makes future enchancements easy to accomplish. The first version, the MC68000, implements only that subset of the complete 68000 architecture that is allowed by current technology constraints. The 68000's design specifies several features, such as floating-point and string operations, that are not

implemented in the first version but have now been specified. Unused space has been left in the architecture to accomodate new features that future advances in technology will make possible (21:44).

Signal Description

The 68000 is packaged in a 64-pin DIP (dual in-line package) as illustrated in Figure III-1. The 64-pin count is significantly greater than the conventional 40-pin microprocessors. To achieve greater data transfer rates, the 68000 does not multiplex its address and data lines as is commonly done. The 64 pins can be functionally broken down into the following groups: 23 for the address bus, 16 for the data bus, five for asynchronous bus control, three for bus arbitration, three for interrupt control, three to indicate the processor state, three for system control, and three for MC6800 peripheral control. The remaining five are used to provide power supply, ground, and the system clock (Figure III-2). Although the internal data paths are 32 bits wide, packaging limitations constrain the number of data pins to 16 and additional operations are required to transfer more than a 16-bit word. The 23 address lines (A1-A23) enable the 68000 to directly address eight megawords of memory space. Individual bytes are addressed via two control lines: the upper and lower data strobes (UDS' and LDS') (signals followed by a "'" are active low). When UDS' is low,

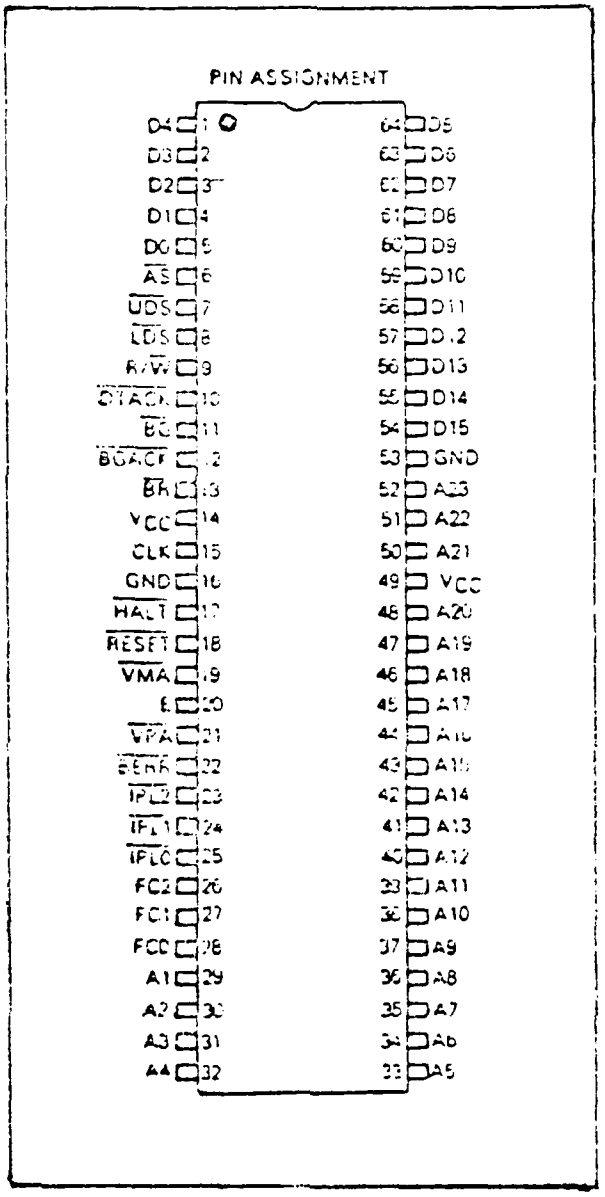


Figure III-1. M68000 Pin Out Diagram (1:1)

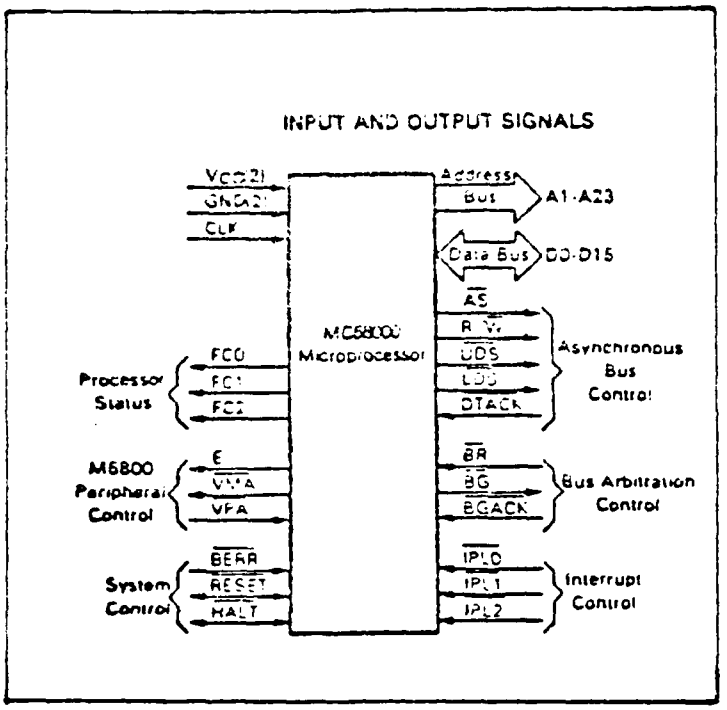


Figure III-2. Input and Output Signals (11:33)

data is transferred on lines D8-D15 of the data bus. If LDS' is low, then data is transferred on lines D0-D7. Finally, if both UDS' and LDS' are low, data is transferred on all 16 data lines.

The 68000 can be interfaced with either asynchronous or synchronous devices and has a separate set of control lines for each. It has three control lines to interface with synchronous peripheral devices in the MC6800 family. They are: enable (E), valid peripheral address (VPA'), and valid memory address (VMA'). The address strobe (AS'), read/write control (R/W), data transfer acknowledge (DTACK'), as well as UDS' and LDS' are used to communicate with asynchronous devices.

Three function code lines (FC0, FC1, and FC2) inform external devices whether the 68000 is in a user or supervisor state. They also indicate the type of cycle currently being executed. An external device such as a memory management unit (MMU) can use these signals to ensure that its operations are conducted when the 68000 is in the proper state. These function control lines can also be decoded to extend the 68000's memory space from 16 megabytes up to 64 megabytes (22:260).

The system control lines are used to halt or reset the processor as well as inform the 68000 of bus errors. The three interrupt control pins carry the priority level of a device requesting interrupt service.

Register Organization (22:229-232)

The MC60000 provides 17 32-bit general-purpose registers, a 32-bit program counter, and a 16-bit status register as illustrated in Figure III-3. Eight of the general-purpose registers are data registers, seven are address registers, and two are system stack pointers (user and supervisor). The eight data registers can be used to perform byte, 16-bit word, or 32-bit longword operations. When a data register is used as either a source or destination operand, only the appropriate low-order portion is changed, the remaining high-order portion is left unchanged. All data registers can also function as index registers under programmer control.

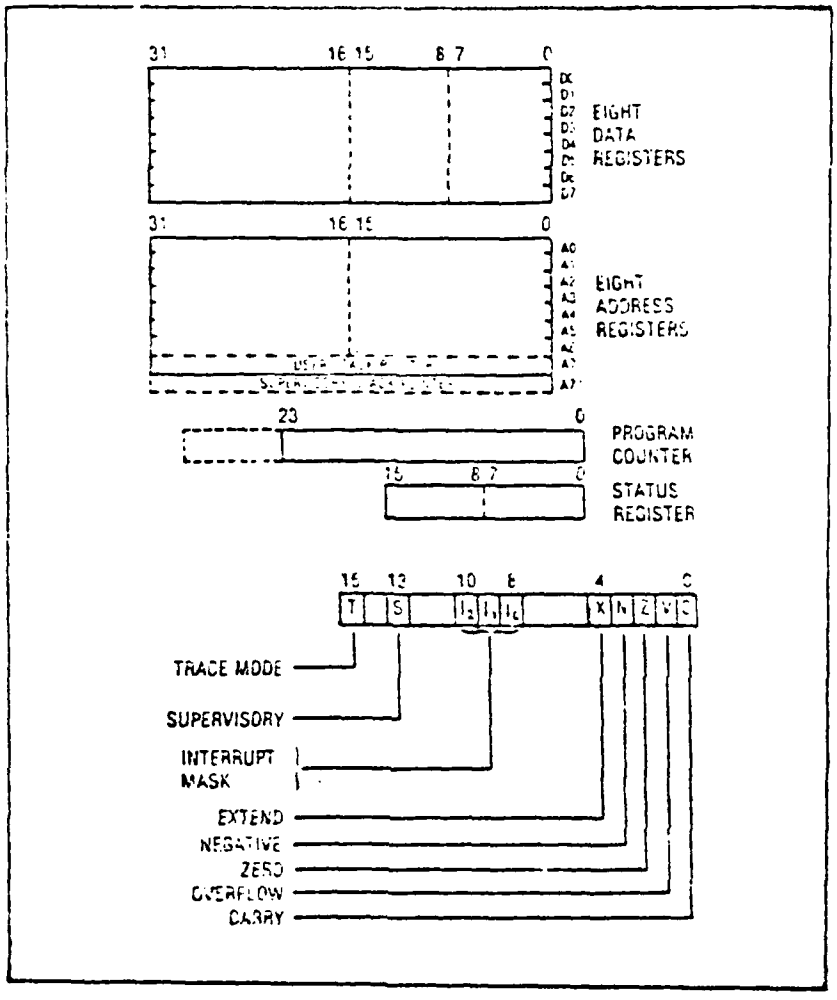


Figure III-3. Register Organization (21:45)

The seven address registers are normally used for word or longword address operations. However, all can also function either as base registers, index registers, or software stack pointers. The address registers do not support byte operations. When an address register is a source of an operand, either the entire low-order word or the entire longword is used depending on the operation size. When used as the destination of an operand, the entire register is affected, regardless of the operation size. If the

operation size is a word, the information destined for the register will automatically be sign extended.

Although the program counter is 32 bits long, only the low-order 24 bits are currently being used. The high-order byte is ignored. Bits 1-23 of the PC are routed to chip's 23 address lines. Bit 0 of the PC is internally encoded with the operand length in the instruction being executed to generate the two data strobe signals UDS' and LDS' described earlier.

The two independent system stack pointers share address A7. The A7 address register acts as a user stack pointer when the 68000 is in a user state, and as a sytem stack pointer when the 68000 is in a supervisor state.

The status register is divided into a system byte and a user byte. The user byte contains five condition code bits (0-4) to record the status of completed operations. They are: carry (C), overflow (V), zero (Z), negative (N), and extend (X). The extend bit acts as a carry for multiprecision arithmetic operations. The status register's system byte has three fields. The interrupt mask is contained in bits 8-10 and provides eight levels of interrupts. With the exception of level seven, all interrupt levels less than or equal to the mask are ignored. Critical interrupts such as system power failures are assigned level seven. The supervisor (S) bit is used to determine whether the 68000 is in a user or supervisor state. The trace mode (T) bit will allow the 68000 to

single step through a program. After each instruction is executed, the 68000 will vector to a special user-written routine that examines the contents of a memory location, register, or performs other debug operations.

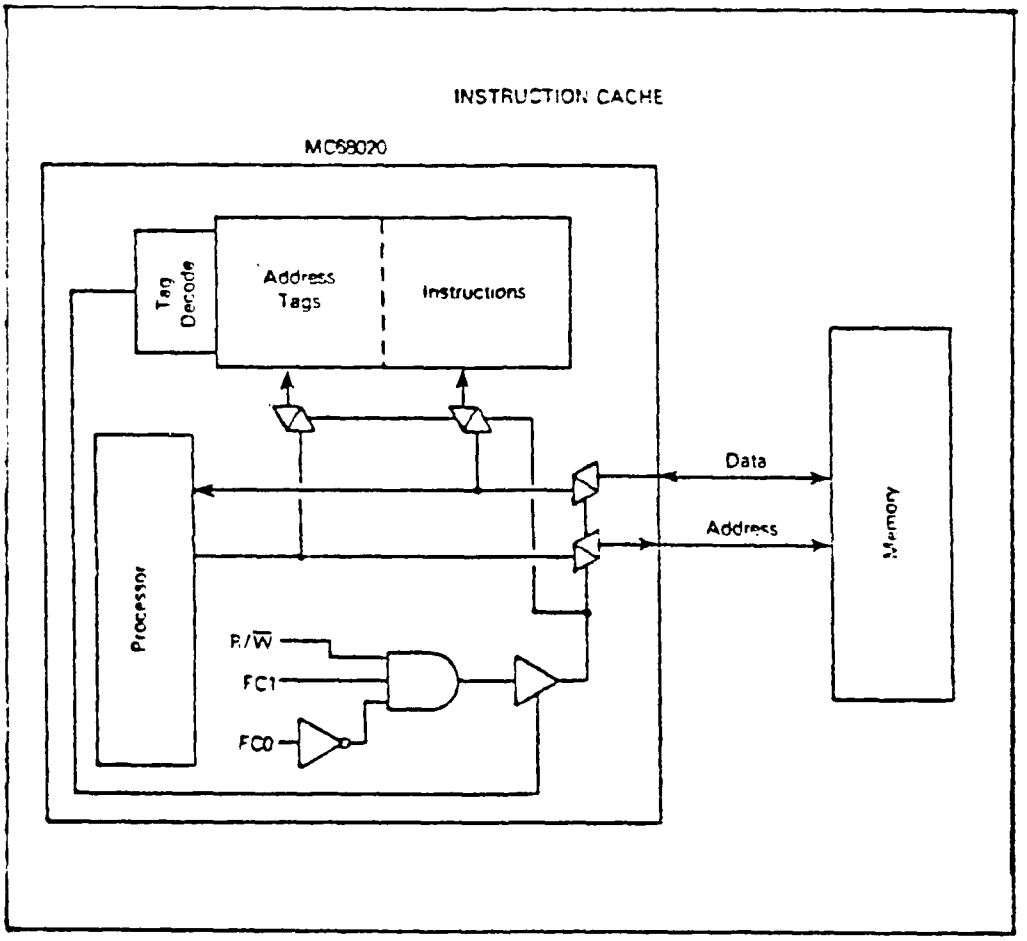


Figure III-4. Instruction Cache (12:3)

System Architecture

The 68000 employs a pipelined architecture in which the instruction fetch, decode, and execute cycles are fully overlapped. An attempt has also been made to minimize delays in branching by prefetching instructions associated

with the most likely branch condition (23:29). Additionally, the MC68020 contains an on-board instruction cache that allows repeated instruction streams to execute significantly faster while freeing the external bus for other processors (Figure III-4).

Pipelined processing is accomplished via the three-sectioned Execution Unit (Figure III-5). Each section contains in its register file some of the 17 general-purpose registers described earlier, as well as others transparent to the user. Each section also contains its own 16-bit ALU. These three sections are dynamically configured by the microcode (they can be isolated or concatenated as necessary) to provide simultaneous address and data processing (15:37).

Instructions are brought in through the 16 data lines into the Instruction Register and Instruction Decode Unit. The Instruction Decode Unit generates an address for the microinstruction in the Micro Control Store. Timing-independent information is sent directly to the Execution Unit.

The Control Store is a two-level structure containing a vertically-microcoded Micro Control Store and a horizontally microcoded Nano Control Store. The Micro Control Store generates a 9-bit address for the nanoinstruction in the Nano Control Store, as well as issues branching signals to the Instruction Decode Unit to cause its next address to be incremented or altered based on conditions received from

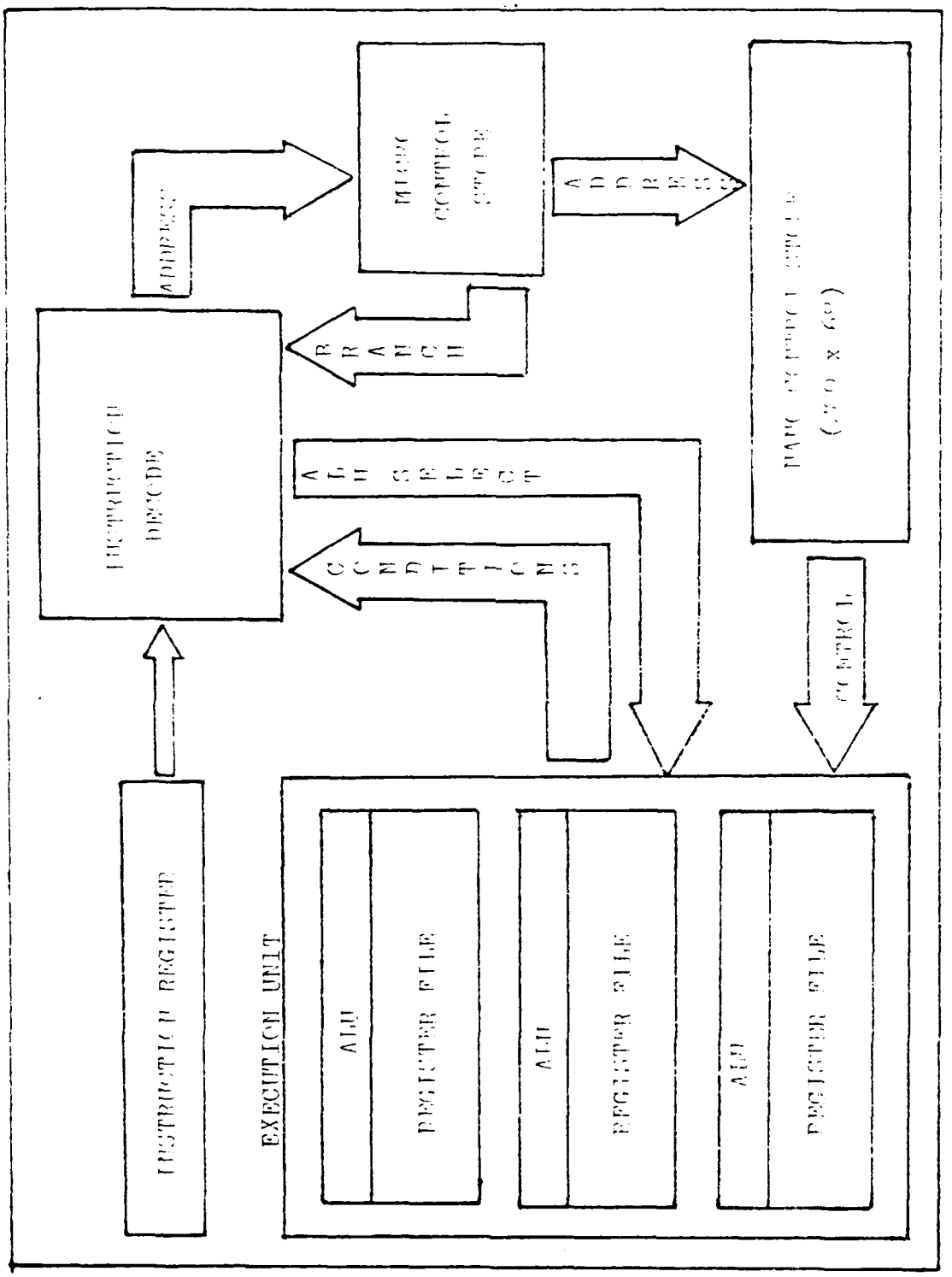


Figure III-5. MC68000 Architecture (23:9)

the Execution Unit. The Nano Control Store houses 280 68-bit control words that directly control the Execution Unit. Approximately 2880 bytes of control store is used, about half that of an equivalent single level implementation (23:29).

The 68000 does not include an on-board memory management unit (MMU). It can be operated with or without one. However, the MC68451 MMU can be interfaced with the 68000 to provide for vertical addressing, segmentation, and memory protection for multiprocessing environments.

Data Types and Organization

The 68000 can operate on five basic data types: bits, bytes, BCD digits, 16-bit words, and 32-bit longwords. These data types are stored in memory as depicted in Figure III-6. Bytes are individually accessible. The high-order byte is assigned its word's even address (Figure III-7) while the low-order byte has an odd address that is one more than its word's. Instructions and data are accessed only on

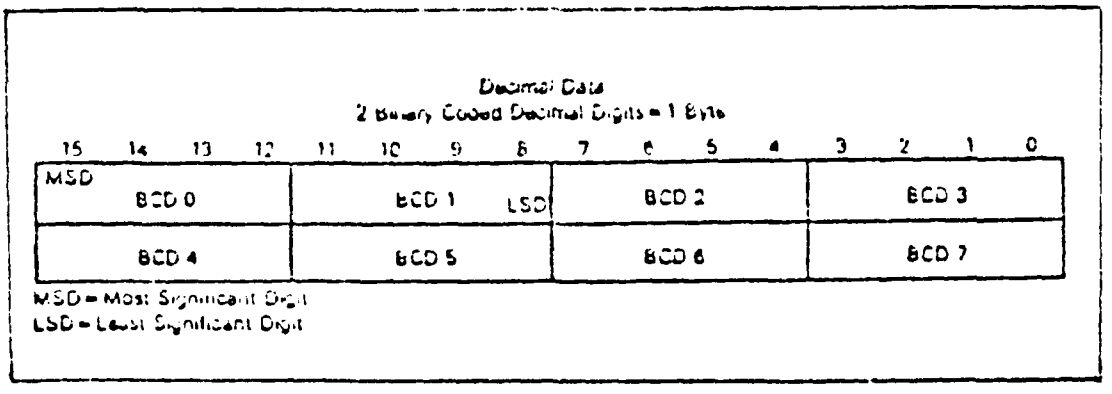


Figure III-6. Data Organization In Memory (10:2-2,2-3)

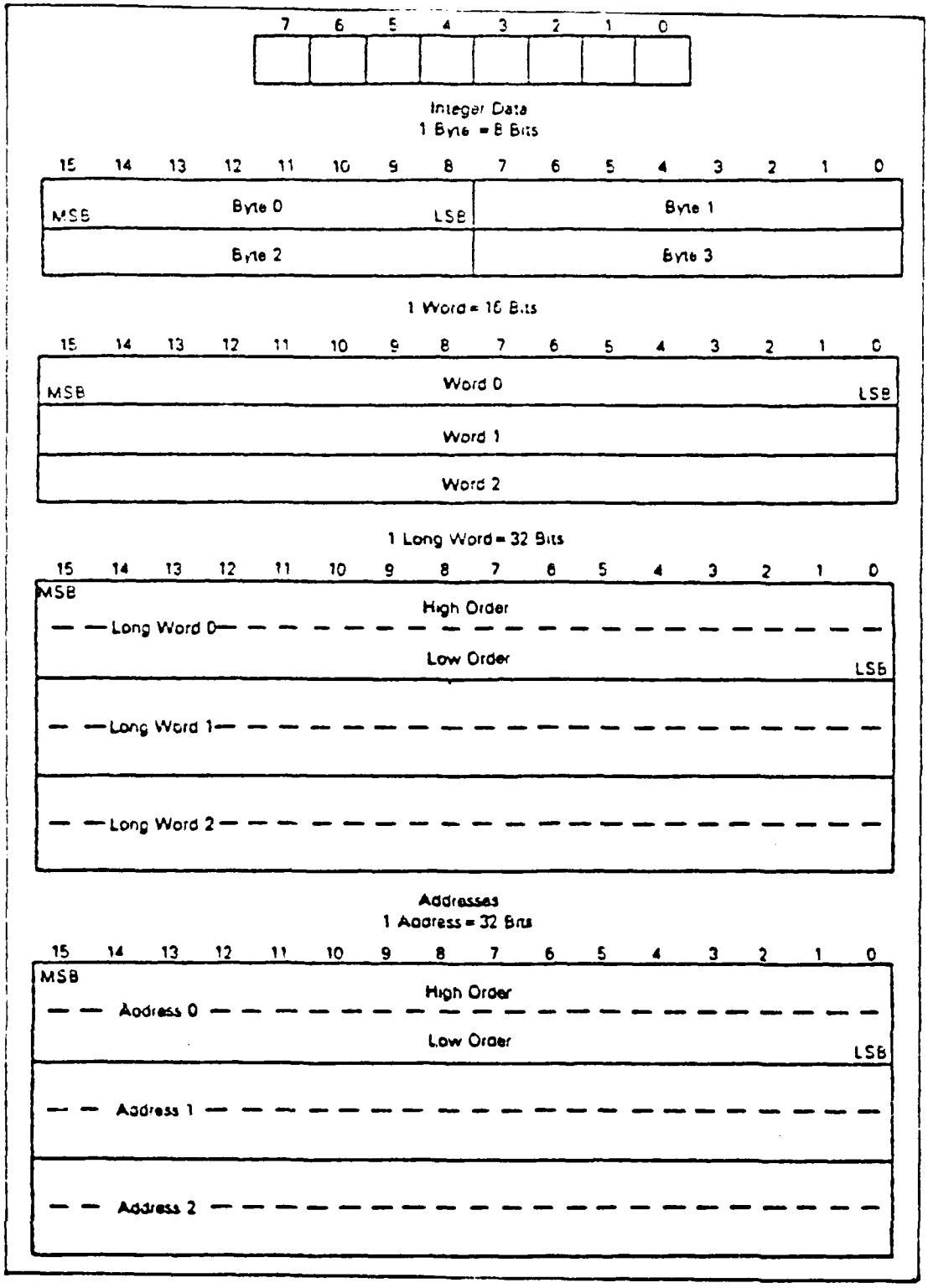


Figure III-6. Data Organization In Memory (continued)

even byte boundaries. Longword data occupies two consecutive addresses in memory (1:2).

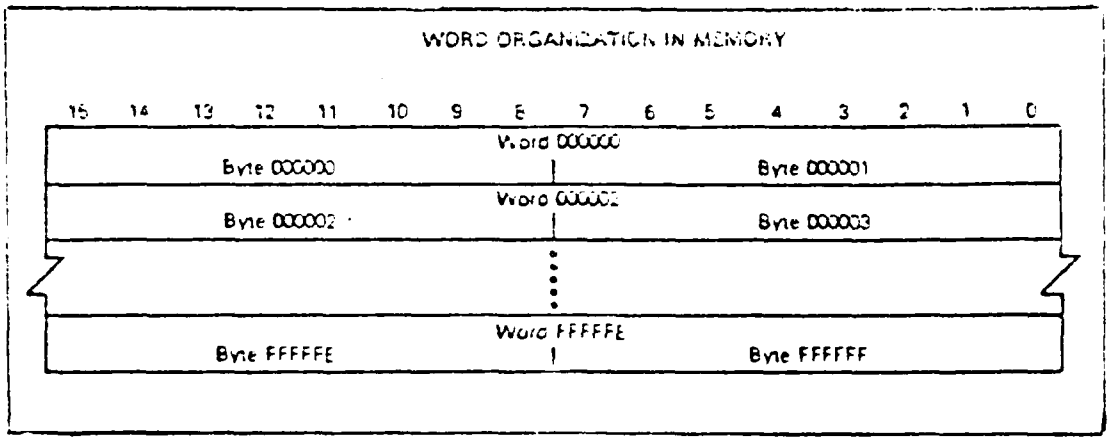


Figure III-7. Memory Word Organization (11:14)

Addressing Modes (20:232-257)

The 68000 offers 14 operand addressing modes giving it a very flexible addressing capability. As Figure III-8 illustrates, these modes fall into six basic groups: register direct, address register indirect, absolute, program counter relative, immediate, and implied.

The register direct mode indicates that the instruction operand is one of the 68000's 17 general-purpose registers. Data or address register direct specifies that the operand is in one of the eight data registers or eight address registers (including the stack pointer) respectively.

The memory address mode indicates that the instruction operand is located in one of the 68000's memory locations. With address register indirect addressing, the contents of an address register points to an operand. There are five

Data Addressing Modes

Mode	Generation
Register Direct Addressing	
Data Register Direct	EA = Dn
Address Register Direct	EA = An
Absolute Data Addressing	
Absolute Short	EA = (Next Word)
Absolute Long	EA = (Next Two Words)
Program Counter Relative Addressing	
Relative with Offset	EA = (PC) + d_{16}
Relative with Index and Offset	EA = (PC) + (Xn) + d_8
Register Indirect Addressing	
Register Indirect	EA = (An)
Postincrement Register Indirect	EA = (An), An ← An + N
Predecrement Register Indirect	An ← An - N, EA = (An)
Register Indirect With Offset	EA = (An) + d_{16}
Indexed Register Indirect With Offset	EA = (An) + (Xn) + d_8
Immediate Data Addressing	
Immediate	DATA = Next Word(s)
Quick Immediate	Inherent Data
Implied Addressing	
Implied Register	EA = SR, USP, SP, PC

NOTES:

EA = Effective Address
An = Address Register
Dn = Data Register
Xn = Address or Data Register used as Index Register
SR = Status Register
PC = Program Counter
d_8 = Eight-bit Offset (displacement)
d_{16} = Sixteen-bit Offset (displacement)
N = 1 for Byte, 2 for Words and 4 for Long Words
( ) = Contents of
← = Replaces

Figure III-8. Addressing Modes (10:1-5)

variations of this mode. In the simplest mode, register indirect, the address register itself holds the effective address. The postincrement and predecrement modes automatically update an address register so that the programmer does not have to use a separate instruction. These modes are useful for moving blocks of data from one section of memory to another. They also permit any address register to be used as a stack pointer so that the programmer is able to maintain eight stacks at once.

Register indirect with offset and indexed register indirect with offset support data table manipulation by permitting offsets and indexes to be applied to an indirect address pointer. Address register indirect with offset adds a 16-bit signed displacement to the contents of an address register as the effective address of an operand. Indexed address register indirect with offset adds an eight-bit signed displacement and the contents of an index register (any one of the address or data registers) to the contents of an address register to produce the effective address of an operand. This mode is useful for accessing two-dimensional arrays.

In absolute addressing, the effective address is contained in the instruction rather than a register. An instruction employing absolute short addressing will contain a 16-bit address whereas an instruction using absolute long addressing will contain a 32-bit address.

Program counter relative addressing modes are useful for developing relocatable programs. In relative with offset, the effective address is the sum of the address in the program counter and a 16-bit displacement. The effective address in relative with index and offset is the sum of the address in the PC, the contents of an index register, and an eight-bit displacement. These two relative modes are useful for manipulating lists, tables, and arrays.

Immediate data addressing is used to specify a constant

data operand as opposed to the contents of a register or memory location. Implicit addressing instructions implicitly refer to the program counter, system stack pointer, user stack pointer, or the status register.

Instruction Set (22:241-257)

Instructions vary from one to five words in length (Figure III-9). All instructions consist of an operation word (op word) containing the instruction type and effective address (addressing mode and register). Additional information may be required to fully specify the operand(s), and this effective address extention is contained in the instruction words that follow the op word. If there are any operands, they vary from a single 16-bit operand to two

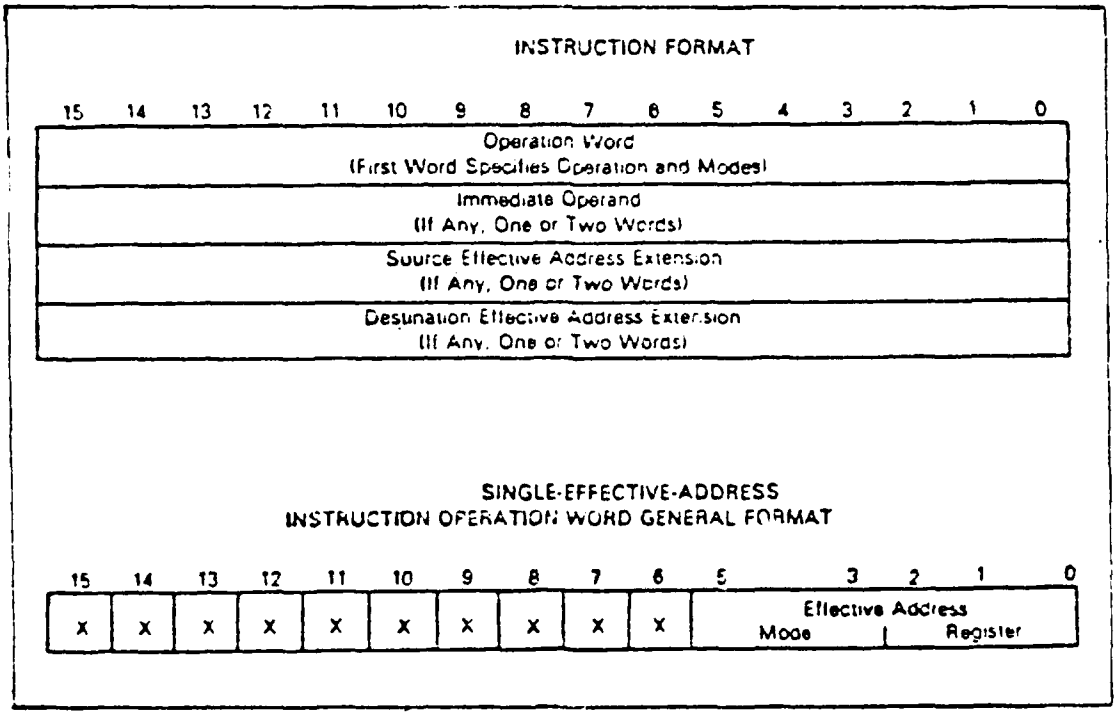


Figure III-9. Instruction Format (10:2-4)

32-bit operands. Because the 68000 was designed by programmers to support programmers, special emphasis has been given to instructions that support high-level languages. The instruction set contains 56 basic instructions, but by combining these with variations of each and tne 14 addressing modes, over 1000 distinct instructions become available.

The instruction set provides for the following operation types: data movement, integer arithmetic, logical, shift and rotate, bit manipulation, Binary Coded Decimal, program control, and system control. Figure III-10 lists the 56 basic instructions. Most instructions can operate on byte, word, or longword data depending on whether the programmer includes a ".B", ".W", or ".L" suffix to the instruction mnemonic.

Data movement instructions are used to transfer information between memory and the general-purpose registers. The principle instruction in this group is the MOVE instruction which can be used to transfer data between memory locations, between a memory location and a data register, or between data registers. EXG will exchange the contents of any two general-purpose registers, and the high-order and low-order 16 bits of a 32-bit register can be exchanged via the SWAP instruction. The 68000's LINK and UNLK instructions are used to allocate and deallocate data areas in the system stack for nested subroutines, linked lists, and other procedures.

Mnemonic	Description
ABCD	Add Decimal with Extend
ADD	Add
AND	Logical And
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
Bcc	Branch Conditionally
BCHG	Bit Test and Change
BCLR	Bit Test and Clear
BRA	Branch Always
BSET	Bit Test and Set
BSR	Branch to Subroutine
BTST	Bit Test
CHK	Check Register Against Bounds
CLR	Clear Operand
CMP	Compare
DBcc	Test Cond., Decrement and Branch
DIVS	Signed Divide
DIVU	Unsigned Divide
EOR	Exclusive Or
EXG	Exchange Registers
EXT	Sign Extend
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link Stack
LSL	Logical Shift Left
LSR	Logical Shift Right
MOVE	Move
MOVEM	Move Multiple Registers
MOVEP	Move Peripheral Data
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD	Negate Decimal with Extend
NEG	Negate
NOP	No Operation
NOT	One's Complement
OR	Logical Or
PEA	Push Effective Address

Figure III-10. MC68000 Instruction Set (10:1-6)

Mnemonic	Description
SBCD	Subtract Decimal with Extend
Scc	Set Conditional
STOP	Stop
SUB	Subtract
SWAP	Swap Data Register Halves
TAS	Test and Set Operand
TRAP	Trap
TRAPV	Trap on Overflow
TST	Test
UNLK	Unlink
RESET	Reset External Devices
ROL	Rotate Left without Extend
ROR	Rotate Right without Extend
ROXL	Rotate Left with Extend
ROXR	Rotate Right with Extend
RTE	Return from Exception
RTR	Return and Restore
RTS	Return from Subroutine

Figure III-10. MC68000 Instruction Set (continued)

Using its integer arithmetic instructions, the 68000 can add, subtract, multiply, divide, and compare two operands. It can also clear, test, sign extend, and negate (two's complement) a single operand. The 68000 also has special instructions to add, subtract, or negate multiprecision numbers (ADDX, SUBX, and NEGX). It is also possible to operate on mixed size data using the sign extend (EXT) instruction. This instruction extends the sign bit as necessary from a byte to a word, or from a word to a longword. Thus, a byte can be added to a word, or a word can be multiplied by a byte.

Multiprecision arithmetic operations on Binary Coded Decimal numbers can be accomplished with the add decimal

with extend (ABCD), subtract decimal with extend (SBCD), and negate decimal with extend (NBCD) instructions.

The 68000 has a capable set of bit-manipulating instructions. It uses four special instructions to test the state of a bit in a memory location or register, record the state of that bit in the zero (Z) condition code flag, and then perform some operation based on the test result. They are: bit test (BTST), bit test and set (BSET), bit test and clear (BCLR), and bit test and change (BCHG).

Program control instructions transfer program control from one portion of a program to another. Of these, the test condition, decrement, and branch (DBcc) is a unique high-level type instruction designed to act as a terminator for repetitive loops. When a DBcc instruction is executed, the 68000 examines the status register condition codes. If a condition is met, program execution falls through to the next instruction. If the condition is not met, the 68000 decrements the specified register. If the register is decremented to -1, program execution falls through to the next instruction; else the 68000 branches to the specified label.

The system control instructions include a trap-generating instruction that initiates a trap operation unconditionally (TRAP), and two instructions that initiate trap instructions based on some condition, trap on overflow (TRAPV) and check register against bounds (CHK). The TRAP instruction can be used for emulating instructions that will

eventually be microcoded in future versions of the 68000.

All instructions can be executed while the 68000 is in the supervisor state. When in the user state, instructions that can have an adverse effect on the system cannot be executed. These include the STOP and RESET instructions, instructions to modify the entire status register, and the move to and from user stack pointer instructions (MOVE USP and MOVE from USP).

The fastest instruction, a register-to-register transfer, executes in four clock cycles, or 400 ns at 10 MHz. The slowest instruction, a signed divide, requires 170 clock cycles or 17 us at 10 MHz.

Because the 68000 uses memory-mapped I/O, there are no separate I/O instructions. Each device is assigned locations in the 68000's memory space and I/O operations are accomplished via the MOVE instructions.

Additionally, floating point and string manipulation instructions are not available in the 68000's instruction set. They have been specified in the design but not implemented in current versions. However, each is presently being implemented either by software or hardware. Two 68000 instruction op codes (1010 and 1111) have been reserved for unimplemented instructions and are assigned Trap vectors for emulation. A user-written routine can accomplish the desired instruction. When newer versions of the 68000 are produced containing the desired instruction, it can be installed and the user-written routines discarded (3:98).

Motorola has also produced support chips that provide floating point operations. The MC68341 ROM and the MC68881 Floating Point Co-processor perform normal arithmetic operations as well as some other related operations (square root, compare, absolute value, etc.) using a floating point format.

IV. Introduction to N.mPc

Introduction

N.mPc is a register transfer level (RTL) simulation system used to assist in the architectural design of digital systems. N.mPc enables system architects, digital engineers, and programmers to test and evaluate their designs prior to system implementation.

N.mPc (PMS notation for "network of microprocessors") was designed and implemented by the Department of Computer Engineering and Science at Case Western Reserve University between 1975 and 1979. Its objectives were to:

1) allow specification of heterogeneous multiprocessor systems;

2) allow modeling at multiple levels of abtraction;

3) allow changes to topologies and microprocessor descriptions with a minimum of work and expense;

4) not impose any particular design style;

5) include facilities for monitoring and controlling simulations of the target achitectures;

6) be useable by non-hardware specialists; and

7) perform well when simulating and evaluating large architectures (18:1).

The resulting system was written in the programming language C and runs on DEC PDP-11 and VAX computers under the VMS and UNIX (V6,V7, and 4.1 BSD) operating systems. N.mPc consists of six major components: the ISP' compiler,

the Metamicro assembler, the Linking/Loader, the Ecologist, the Simulated Memory Processor (SMP), and the Runtime system. They combine to create and control target architecture simulations. The system hardware to be simulated is described by these three components:

1) ISP'. ISP' (ISP' is an extention of the Instruction Set Processor language developed by Bell and Newell) is a RTL compiler that includes many features of high level languages to allow the user to model system hardware components. The ISP' compiler translates ISP' hardware descriptions into executable object modules for the host computer.

2) Ecologist. The Ecologist defines the structure of the target system. It uses a system topology file describing the ISP' object modules to be combined to form the simulation program.

3) SMP. The SMP initializes the target machine's memory components with the programs developed by the user to be hosted on the simulated system.

The two components used to develop software for the simulation model are:

1) Metamicro. Metamicro is a generalized assembler that allows the user to develop an assembler for any target processor by describing it through a macro-based language (19:3). It allows the user to specify the format, mnemonics, and associated bit patterns of the target instruction set (18:3) so that an assembly language program developed by the

user can execute on an ISP' hardware engine (5:77).

2) Linking/Loader. The Linking/Loader enables the user to develop a linker and loader for any target processor by describing its addressing modes. It links modules assembled separately by Metamicro and loads the resulting object code into the simulated memory in accordance with strategies developed by the user.

The system designer interacts with the simulation program created by the Ecologist through the Runtime system. The Runtime system allows the user to control and monitor the simulation, as well as create the performance-evaluation and simulation libraries (5:77). With the Runtime system, the user is able to gather the same performance information that would ordinarily require the use of logic analyzers, oscilloscopes, and program debuggers.

Figure IV-1 on the following page presents a detailed block diagram of the N.mPc system and enhances the narrative that follows. To simulate a system design with N.mPc, the individual hardware components are first described using ISP'. The Ecologist then uses a description of the interconnection topology to bind these compiled hardware descriptions into a network of communicating processors. Assemblers for the processors comprising the system and their associated application programs are then developed using Metamicro. Linking/Loader is used to link the various modules assembled by Metamicro and then load the resulting

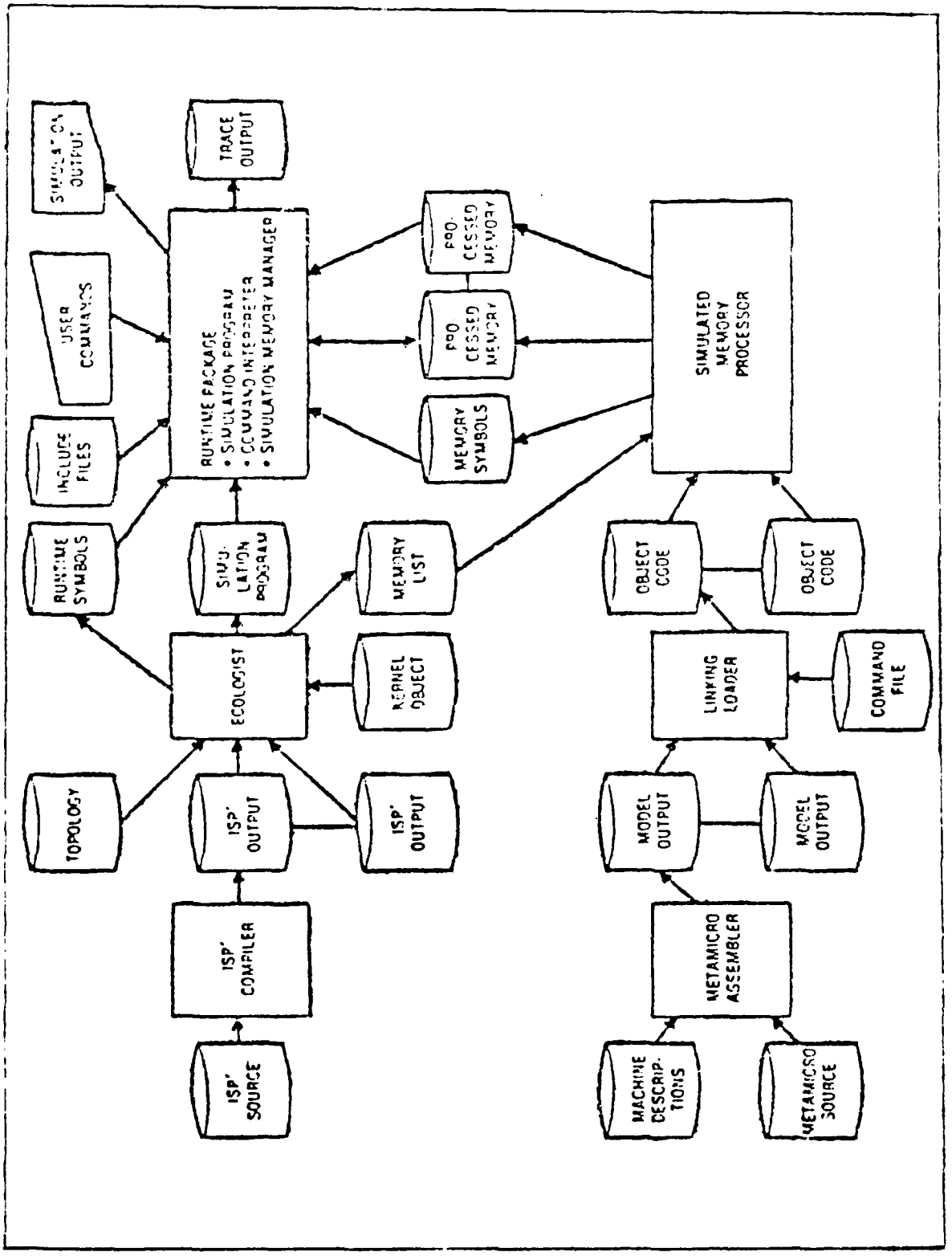


Figure IV-1. N.mPc Block Diagram (18:47)

object code into the memory components of the simulation model. Finally, using the Runtime package, the user can simulate and observe the operation of the hardware and software components to test and evaluate the system.

ISP' (20:1-5)

As mentioned earlier, ISP' is a programming language for describing processors and other hardware elements at the register transfer level. Systems designers use ISP' to model many types of hardware elements such as ALU's, memories, or CPU's. An ISP' source program consists of structure, procedure, and process declarations. The processor's structure is first described through three declaration types:

1) states - the microprocessor's registers are represented by states. Instruction registers, program counters, flags, etc. are declared as states. The following example declares an array of eight registers, each containing 8 bits:

```
state  D[0:7]<7:0>;
```

2) memories- random access memories are declared to host the simulated microprocessor's instructions and data. A memory with 64K 16-bit words can be declared with the following declaration:

```
memory  M[0:65537]<15:0>;
```

3) ports - a microprocessor's address bus, data bus, and

control signals are declared as ports. Ports correspond to the pins on an IC chip. The ports of system hardware elements may be later connected by the Ecologist to enable communications between microprocessors or between a microprocessor and its peripherals. Here is an example of an 8-bit data bus declaration:

```
port  databus<1:8>;
```

The hardware structure declarations are followed by procedure and process declarations. A process is loop whose instructions are repeated for the duration of the simulation. The following example from the ISP' user's manual illustrates a process:

```
state counter<16>;

main := (
        delay(1);
        counter <= counter + 1
        )
```

Main is a keyword which identifies a process. The process "main" and its instructions (enclosed in parenthesis) are separated by the delimiter ":=". In this process, a 16-bit counter is continually incremented with one simulation unit of time delay between incrementations. The actual simulation time units are defined in the system topology description processed by the Ecologist. This counter would operate at eight MHZ if a simulation time unit was defined to be 125 nanoseconds (ns) in the topology file. Note from

the preceding example that ISP' statements are separated by a ";" and assignments are made with a "<=" symbol. Consider the slightly more complex example from the same manual:

```
state  counter <16>, save <16>;
port   ck, switch;

when (ck:lead) := (
                  if switch
                          (counter <= save;
                           save <= counter;
                           next
                          );
                  counter <= counter + 1
                  )
```

In addition to "counter", there exists another 16-bit register "save", and two single-line ports "ck" and "switch". When the leading edge of "ck" occurs, the processor accomplishes one of two possible actions; if switch is true, the registers "counter" and "save" are exchanged and then "counter" is incremented, else "counter" is incremented without a prior register swap. Because ISP' is a register transfer language, assignments are performed concurrently. If not, in the above example the register "save" would receive its old value. The "next" statement enables sequential assignments by causing the preceding assignments to be made. Without it, all three of the example assignments would be made concurrently (if switch were true) and the result in "counter" would be indeterminate. Because the statement "next" is implied at the end of each process, the assignment to "counter" is made at the end of the "when" process. The "delay" and "wait"

statements also cause assignments to be made. For additional detail on these and other constructs, refer to the ISP' User's Manual. To further the reader's "feel" for the language, Appendix K contains an ISP' description of the Reduced Instruction Set Computer (RISC 1) developed at the University of California at Berkeley. This model was developed as a part of a local computer architecture course to advance the understanding of RISC's innovative architectural concepts (e.g., register windowing, constant width instructions, and consistent instruction execution times).

Metamicro (16)

Metamicro is a generalized micro assembler which uses a description of a processor's instruction set to assemble programs. Rogers and Ordy characterize an assembler as a translator which takes a computer instruction in a form understandable by its writer, and then creates an instruction with the same meaning, and in a form understood by the digital hardware (16:17). Metamicro satisfies this description by allowing the user to specify the input form, output form, and translation rules for a given microprocessor's assembly language. The user first describes the construction of the target processor's instruction set using Metamicro constructs. This description is then used to assemble applications programs written in the target processor's instruction set. Figure IV-2 contains a partial Metamicro description of the Intel

8080 instruction set. Even though only two instructions are modeled (Add Register to A (add) and Add Register to A with Carry (adc)), it will serve as an adequate introduction to Metamicro.

```
instr    inst[3,1]<8> $ ! three words of eight bits each
                        ! default instruction length one

format   op      = inst[0]<7:6>, !    op code
         dst     = inst[0]<5:3>, !    destruction
         scr     = inst[0]<2:0>$ !    source

macro    b       = 0  &,
         c       = 1  &,
         d       = 2  &,
         e       = 3  &,
         h       = 4  &,
         l       = 5  &,
         m       = 6  &,
         a       = 7  &,

      sreg(x) = src = x $ &,

      add(x)  = op  = 2;
                dst = 0;
                sreg(x) & ,        ! add reg or mem
      adc(x)  = op  = 2;
                dst = 0;
                sreg(x) & $        ! adc reg or mem
```

Figure IV-2. Metamicro Description (19:Chap 2:7)

The instruction declaration is used to inform the assembler of the instruction's size and format. It allows the user to specify both the maximum and mimimum number of words in an instruction, as well as its word width. In the example program above, the instruction declaration is:

```
instr        inst[3,1]<8> $
```

The name "inst" is used to symbolically reference the instruction. To enable variable length instructions, the maximum instruction word size has been set at three while the minimum is declared to be one ([3,1]). Since the basic word size has been declared to be eight bits (<8>), we can create one, two, or three byte instructions. The $ character terminates all Metamicro statements. The "!" is used for commenting by causing Metamicro to ignore the remainder of the current line.

The format declaration specifies the subfields of each instruction word signficant to the model that we wish to symbolically reference (19:Chap 2:8). In the example format declaration:

```
format    op   =   inst[0] <7:6>,   ! op  code
          dst  =   inst[0] <5:3>,   ! destination
          src  =   inst[0] <2:0>$   ! source
```

the instruction bit fields that contain the op code and the source and destination registers are identified. Specifically, the src register is identified by bits 0-2 in the first byte of the instruction and the dst register is identified by bits 3-5. Finally, the op code is contained in bits 6-7.

A macro is used to translate the instruction mnemonics developed by the user to correspond with a microprocessor's assembler into statements compatible with Metamicro. A macro element has the following structure:

```
macro_name = macro_body &
```

Macro_name is the identifier by which the macro will be referenced. A parenthesized parameter list may be appended to macro_name. The character & is used to delineate multiple macros within the macro declaration section. Macro_body may contain statements, macro calls, etc.

In the example of Figure IV-2, the macro declaration section begins with the keyword "macro" and forms the bulk of the instruction set description.

```
macro b = 0 &,
      c = 1 &,
      d = 2 &,
      e = 3 &,
      h = 4 &,
      l = 5 &,
      m = 6 &,
      a = 7 &,

      sreg(x) = src = x $ &,

      add(x) = op = 2;
               dst = 0;
               sreg(x) &,  ! add reg or mem

      adc(x) = op = 2;
               dst = 0;
               sreg(x) &$  ! adc reg or mem
```

The macros b,c,d,e,h,l,m, and a provide values for the respective register names which are assigned to the src subfield by another macro "sreg". The macro sreg is a utility macro that simplifies the coding of statements which appear in several macros. It makes assignments to the src field of an instruction.

Before continuing, it is necessary to introduce an

applications program that can be assembled by the above Metamicro description (19:Chap 2:8).

```
include 8080.m$
begin
  add(b)
  adc(m)
end
```

The characteristics of the processor are defined in the declaration section. This file comprises the corresponding instruction section of the example Metamicro program of Figure IV-2. It contains the applications program instructions that are assembled according to the translation rules established in the declaration section.

The instruction section is separated from the declaration section by the keyword "begin" and terminated by the keyword "end". In this example, the declaration section has been put in a file of its own (8080.m) and the user begins the source code with an "include" statement to prepend it to the instruction section.

Whenever one of the two instruction mnemonics (add or adc) is encountered by Metamicro, its corresponding macro in the declaration section is expanded in line during the assembly process so that the correct assignments are made to instruction subfields. For example, when the add(b) instruction is encountered, the add(x) macro assigns 2 (10b) to the op code subfield, 0 (000b) as the destination register, and, through the invocation of macros b and sreg(x), also assigns 0 (000b) to the source register. The

adc(m) instruction differs only in that register six identifies the memory location to be added to register zero. The macro sreg(x) is common to both instructions.

A Metamicro description of the RISC 1's instruction set is contained in Appendix L to give the reader an opportunity to amplify this brief introduction to Metamicro. This assembler will transform programs written in RISC's assembly language into machine code that can be executed by the RISC 1 processor modeled in Appendix K.

Linking/Loader (17)

The user can develop a linker and loader for any target processor by describing its addressing modes with the Linking/Loader. The Linking/Loader links the various files assembled by Metamicro and loads the resulting object code into the simulated memory. Options are available to specify different loading algorithms that may be more suitable for a given simulation.

To generate the actual executable instructions, the user builds a command program to describe to Linking/Loader how instructions are modified to resolve label references made in Metamicro. A Linking/Loader command program is constructed from five declaration types:

1) instr - informs the Linking/Loader of an instruction's size and format;

2) format - specifies the instruction subfields that will be symbolically referenced;

3) mode - describes how referenced labels are resolved

into address operands;

4) space - declares available memory space; and

5) transfer - provides the Linking/Loader with information pertinent to the relocation of instruction segments.

Figure IV-3 is a generalized Linking/Loader command program for the Intel 8080 microprocessor.

```
instr     inst[3,1]<8>$

format    op  = inst[0]<7:6>,
          dst = inst[0]<5:3>,
          src = inst[0]<2:0>,
          rx  = inst[0]<5:4>,
          wd1 = inst[0]<7:0>,
          wd2 = inst[1]<7:0>,
          wd3 = inst[2]<7:0>$

space <0:4095>$

mode      case length eql 3:
               wd2 = address$
               wd3 = address^-8$
          break$
          esac,
          default:
               wd1 = address$
               wd2 = address^-8$
          break$
          esac$

 transfer   {      new
                   wd1 = 0303$
                   wd2 = address$
                   wd3 = address^-8$
                   length = 3$      }
```

Figure IV-3. Linking/loader Command Program (19:Chap 2:10)

The Linking/Loader "instr" and "format" declarations are

equivalent to those of Metamicro that were presented earlier and will not be discussed again here.

The "mode" declaration details how addresses are resolved for a particular microprocessor. The mode declaration section processes only those instructions that reference labels. When Metamicro builds an instruction it tracks the number of labels referenced by that instruction and places an associated address for each reference into an address array for that instruction.

The mode declaration is similar to the case statement of several high level programming languages. The algorithm of the mode declaration of Figure IV-3 first determines the addressing mode of the instruction by examining its length. The 8080 has a single addressing mode, direct address, and it occurs when the instruction is three bytes long (19:Chap2:10). Each instruction generated by Metamirco has a length associated with it and is stored in the variable "length". If this initial expression is true, the second word of the instruction (wd2) receives the lower eight bits of the label's address and the instruction's third word (wd3) receives the upper eight bits. The "=" means to logically "or" the expression value on the right into the identifier on the left. If the expression on the right exceeds the bit length of the identifier on the left, then only the least significant bits are or'ed. Because "address" is a 32-bit field and wd2 is eight, wd2 receives the eight least significant bits of address. Note that

"address" is the first element in the address array created by Metamicro for an instruction referencing a label. The "^" is the shift operator and the value to its right is the shift value. A positive shift value signifies a left logical shift by abs(value) bits whereas a negative value indicates a right arithmetic shift by abs(value) bits. In the example of Figure IV-3, wd3 receives bits 8-15 of "address" as a result of the "shift and or" operation. This technique is frequently used to break large addresses into smaller instruction words.

The break statement causes the current case statement to be exited without executing any more statements in the mode declaration. The current instruction is thus resolved and placed in the output file. Case statements are terminated by the keyword "esac".

If the initial case entrance expression evaluated to false, then the default case is entered and words one and two of the instruction (wd1 and wd2) would receive the address. This is the case when the label referred to is a data constant (19:Chap2:11).

The space declaration describes the target processor's memory space. Linking/Loader allocates applications program instructions assembled by Metamicro into this space. The space declaration of Figure IV-3 defines a 4k memory.

During the loading process, the Linking/Loader may break a group of logically contiguous instructions into segments and place them into disjoint areas of the target

machine's defined memory space. To ensure that target machine instructions appear logically contiguous to the user, when this occurs the Linking/Loader adds a new statement to unconditionally transfer the program flow to the disjoint segments. The Linking/Loader builds this new instruction in accordance with the format specified in the transfer declaration.

Linking/Loader places the transfer destination address into the variable "address" and the user must use this variable to include the transfer address into the unconditional transfer instruction. In the example of Figure IV-3, the Unconditional Jump instruction is used to provide logical continuity if memory allocation is not physically contiguous (19:Chap2:11).

Ecologist (13)

The Ecologist uses the files representing the descriptions of the system's hardware and software components to build the N.mPc simulations. The Ecologist builds the simulation from a topology file constructed by the user to describe the interconnections between the system components. The topology file describes the total system to be modeled. Each hardware component of the sytem described by ISP' models must be compiled before the Ecologist can build the simulation. If the simulation includes memories, the applications programs must also have been assembled by Metamicro and then linked and loaded by the Linking/Loader.

The topology file is comprised of five declaration

sections which may or may not be included depending on the nature of the ISP' model.

1) Signal - A signal is the name of the connection that exists between ISP' ports. The value of a signal is the logical "or" of all the ports tied to it. An example of a signal declaration is:

```
signal  ADDRESS(23), DATA(16), R_W;
```

This declaration begins with the keyword "signal" and describes a 23-bit address bus, 16-bit data bus, and a read/write control line.

2) Processor - For each ISP' output file comprising the simulation, a processor declaration must exist. The following example depicts a processor declaration.

```
processor     cpu = "M68000.sim";
```

The keyword "processor" begins the process declaration. At simulation time, the ISP' hardware model will be refered to by the name "cpu". M68000.sim is the UNIX file containing the ISP' compiled output.

3) Time Delay - If an ISP' model has used a timed delay call, the Ecologist will expect a time delay declaration for that module. The time delay declaration is used to give the relative delay times a real time analogy. The basic unit of simulation time is one ns. An example of the time delay declaration is:

```
time delay 60 ns;
```

This declaration will cause each unit of delay in the ISP' model to correspond to 60 ns.

4) Connection - The user connects ISP' module ports to the declared signals via the connection declaration. All ports in an ISP' module must be connected to a signal. For example,

```
connection abus = ADDRESS,
           dbus = DATA,
             rw = R_W;
```

would connect the ports abus, dbus, and rw to the signals declared earlier. Note that ports and signals must be the same width to enable connection.

5) Initial - The initial contents of ISP' memories are specified by the initial declaration. Each memory image produced by Metamicro, Linking/Loader, and SMP action is associated with an ISP' model of the supporting memory via the "initial" declaration. In the following example, the memory image "sortimage" (sort algorithm) is bound to a memory named "mem":

```
initial    mem = sortimage;
```

Figure IV-4 illustrates a topology file that describes a simulation comprised of a Motorola 68000 with an external memory that will be loaded with the sorting algorithm "sortimage" introduced above.

```
signal
      ADDRESS(23),              ! Address Bus
      AS,                       ! Address Strobe
      DATA(16),                 ! Data Bus
      UDS,                      ! Upper Data Strobe
      LDS,                      ! Lower Data Strobe
      DTACK,                    ! Data Transfer Acknowledge
      R_W,                      ! Read/Write
      FC(3);                    ! Function Code

processor     cpu      = "m68000b.sim";
time delay    60 ns;
connections   ADDRESS  = ADDRESS,
              AS       = AS,
              DATA     = DATA,
              UDS      = UDS,
              LDS      = LDS,
              DTACK    = DTACK,
              R_W      = R_W,
              FC       = FC;

processor     mem      = "m68000bm.sim";
time delay    100 ns;
connections   ADDRESS  = ADDRESS,
              AS       = AS,
              DATA     = DATA,
              UDS      = UDS,
              LDS      = LDS,
              DTACK    = DTACK,
              R_W      = R_W,
              FC       = FC
initial       mem      = sortimage;
```

Figure IV-4. Motorola MC68000 Topology File

Simulated memory Processor (13:10,11)

The Simulated Memory Processor (SMP) is responsible for preparing memories for simulation. If the simulation uses memories, the Ecologist collects a list of the memory files specified in the "initial" declaration of the topology file and passes it to the SMP for processing. The SMP's two major functions are memory image processing and global label collection.

The Linking/Loader produces a memory image file

representing the linked output of up to ten Metamicro assembled input programs. SMP takes this file and reformats it into fixed size pages and at the same time produces a page table. The user can specify the page size. Depending on the memory locations addressed, pages are swapped between the simulation program and the Simulated Memory Managers over UNIX pipes.

All labels that are declared as global in Metamicro source programs are placed into a common file by the SMP for use during the simulation. At runtime, these global labels may be used to reference addresses in the memory being simulated.

Runtime (14)

The user executes the simulation through the Runtime package. The Runtime's Command Interpreter(CI) provides the interface between the simulation and the user. This process accepts commands from the user to examine or modify the simulation states, to control the execution of the simulation, to set execution breakpoints, or to establish mechanisms that allow the automatic collection of data from a running simulation (9:4).

The user begins a simulation by entering the simulation program name produced by the Ecologist. After an introductory message, the CI issues a "#" prompt enabling the user to enter a command. Several of these Runtime commands are introduced below. For a complete and more detailed list, refer to the N.mPc Runtime User's Manual.

1) examine - the examine command is used to display the contents of a single state, port, or memory location. For example,

examine cpu:abus

will display the current value of the abus port from the ISP' process cpu. Note that the ISP' process name will be one specified in the topology file.

2) deposit - deposit allows the user to write a value into a state, port, or memory location. For example,

deposit 0b00011011 cpu:ir

will place the binary value 00011011 into cpu's ir register. Note that 0b specifies that the value that follows will be binary.

3) states - while "examine" operates on a single state, the states command allows all states for a given ISP' process to be examined. Example:

states cpu

This command will display the contents of all registers on board the processor "cpu".

4) ports - performs the same function for ports as states does for registers.

5) memory - memory is used to examine multiple memory locations. For example,

memory cpu:mem 100 110

will display locations 100-110 of a memory named "mem" that has been declared in the ISP' process cpu. Note that numbers beginning with 1-9 are assumed to be decimal.

6) display - the display command is used to display the contents of a state, port, or memory location when it is written to during a running simulation. For example,

display cpu:ir

will cause the contents of the register ir to be displayed each time it is written to. Also included in the display will be the current simulation time.

7) bkpt - the bkpt (breakpoint) command causes the simulation to stop when a particular time or condition exists. For example,

bkpt 1250

will cause the simulation to stop in 1250 ns so that the user can monitor the simulation.

8) repeat - if the repeat command prepends bkpt then the breakpoint will be continually repeated. For example,

repeat bkpt 1250

will cause the simulation to stop every 1250 ns.

9) run - the run command starts a simulation executing or restarts a stopped simulation.

10) quit - the command quit terminates a simulation.

Local System Access

Appendix A provides the interested reader with information to supplement departmental N.mPc documentation packages to allow access and use of the system as locally installed. Included is:

1) a functional description of each of N.mPc's major components and simulation files,

2) an organizational representation of each N.mPc component along with their input and output files,

3) instructions for accessing the system,

4) a listing of N.mPc's directory structure that includes its microprocessor library, and

5) an example of a N.mPc simulation output product.

model the MC68000 for simulation.

ISP' could be used to construct an equivalent model of the 68000 basically by ignoring some of its advanced features and instead using its rudimentary operations to create the necessary mechanisms to emulate CDL's control, timing, and parallelism capabilities. Prior to outlining the adjustments necessary to create an equivalent ISP' model from the CDL model provided, these two CHDL's will be compared and contrasted.

CDL/ISP' Comparison

Although both ISP' and CDL are languages capable of describing computer components and hardware operations at the RTL (computer organization and design), there is a *fundamental difference* between the two. CDL is a "nonprocedural" language. Nonprocedural languages attach no meaning to the lexicographical ordering of the statements describing the operation of the system (2:138). Microstatements are associated with a label that describes the conditions in which they are executed. As an example, consider the following CDL execution statements extracted from the MC68000 model (8:VI-17):

```
/ctrl*K(4)*P(1)/  PFR <- EXDBUF
/ctrl*K(4)*P(2)/  ASN <- 1, LDSN <- 1, UDSN <- 1, T <- 0,
                  IR <- PFR, PC <- PCadd2
```

The label /ctrl*K(4)*P(1)/ specifies the conditions in which the microstatement PFR <- EXDBUF is performed. Reordering

these two execution statements would have no effect on the timing of the microoperations. Whenever the conditions of the label are satisfied, its microstatements are executed. In contrast, here is the ISP' equivalent:

```
if ctrl and K eql 4
   (
   wait (P1:lead);
   PFR = EXDBUF;
   next;
   wait (P2:lead);
   ASN = 1;
   LDSN = 1;
   UDSN = 1;
   T = 0;
   IR = PFR;
   PC = PC + 2;
   next
   );
```

Being a "procedural" language, the sequential ordering of ISP' statements implies an explicit ordering of its activities, and the activation of activities is conditioned by the completion of the preceding ones (2:138). If the conditions of the "if" statement are satisfied, then the microstatements enclosed in parenthesis are executed. Also, should the order of the two "wait" statements be changed, then the execution of their following microstatements will be reversed. This is an undesirable result since we want PFR (prefetch register) to receive EXDBUF (external data buffer) before it is loaded into IR (instruction register). In CDL, there is no provision for the partitioning of a hardware description into blocks of related execution statements to reflect a particular organization or hierarchy

of activities (2:144). Execution statements have a sequential appearance with two columns formed by their labels and microstatements (2:138). ISP' descriptions follow the structure of C programs and possesses many of its high-level programming constructs (e.g., case, do-until, while). As such, it does not impose a rigid repetitive design style upon the user. The above example also highlights several other major differences between CDL and ISP'.

Because CDL labels identify the conditions under which their associated microstatements are performed, there is a clear delineation between data and control. Special control variables separated by slashes specify the conditions necessary to execute the microstatements. Sequencing through microstatements is accomplished by modifying the control variables in the label. If the label's control expression evaluates to true, then its accompanying microstatements are executed, otherwise they are ignored. In contrast, ISP's conditional statements form the equivalent of labels. Delinear ambiguity exists between the conditional expressions representing control, and their dependent microstatements. The conditional test is performed and the following microstatements are either executed or skipped depending on the outcome. Additionally, in CDL, timing is provided by including in the label a specialized control component, the clock. The following example from Hamby and Guillory's model declaration section

illustrates clock usage:

```
Clock,  P(1-2)          $ two phase clock
```

This clock statement declares a two-phase clock to provide a two-phase clock cycle for their MC68000 model (4:6). As a result, clock pulses P(1) and P(2) alternate values between one and zero in accordance with a frequency specified by the host simulation package. Action by this independent activity automatically modifies label control expression to direct the timing of microstatement executions. ISP' has no such clock structure. As with control, the timing of its microstatements are provided by conditional statements in conjunction with several specialized monitoring facilities such as the "when" process and "wait" statement. Below is an example of a wait statement taken from the previous ISP' example.

```
wait(P1:lead);
PFR = EXDBUF;
```

In this example, the wait statement will cause the process to halt execution until the port signal "P1" transitions high. The microstatements following it will not be executed until this occurs. Similarly,

```
when(ck:lead) :=
              (
              address = adr_reg;
              read = lo
              )
```

will cause the series of statements associated with the "when" process to be executed when "ck" transitions high.

However, these two monitoring facilities can only be used in conjunction with external signals (port signals) and cannot be triggered by local state change (change in the state of an internal register). This requirement prohibits their use when a single hardware component is modeled.

ISP' also provides a "delay" statement to cause a process to wait a specified number of time units before resuming. The time specified is independent of real time, rather it is a simulated time that corresponds to the real time in the system being simulated. The delay statement is not used to direct the execution of microstatements but is used to specify the simulation time in which a microstatement is executed. Whether a microstatement is executed or not is dependent upon the outcome of prior conditional statements.

Another of the specialized structures available in CDL but not found in ISP' is the "decoder". As does its real word counterpart, CDL's decoder translates the binary value of its inputs into a single output signal (4:6). For example, the MC68000 decoder statement

```
Decoder,   K(0-255) = T(0-7)
```

specifies a 8 x 256 decoder, K. Its eight inputs are attached to register T (which had been previously declared); thus, the binary value of T will determine which of K's 256

output lines becomes high. Decoder output is often used in a label's control expression to direct the sequencing of CDL microstatement execution. Another example taken from the MC68000 CDL model depicts this process.

```
/ctrl*K(1)*P(2)/  T <- CountupT
/ctrl*K(2)*P(1)/  IF (DTACKN = 0) THEN
                  (T <- CountupT)
```

From this example, it is easy to see that the output of decoder K controls the seqencing of these two microstatements. Since the decoder K is attached to register T, when the operator "Countup" in the first microstatement increments register T, K(2) will become high and provide the potential for the next microstatement to be executed. While such a combinatorial circuit is not provided by ISP', it can be duplicated by the following microstatement:

```
K[T]
```

if registers T and K have previously been declared as follows:

```
state  T<8:0>,
       K[0:255];
```

In this example, T has nine bits because the most significant bit is interpreted by ISP' as the sign bit.

A statement found in ISP' but not CDL is the "next" statement. As mentioned in the previous chapter, the next

statement forces sequentiality. All assignments preceding it are performed concurrently. In a CDL execution statement, all microstatements associated with the statement label are executed concurrently. Sequentiality is achieved via decoders and clock pulses in the label's control expression. Similarly, ISP' uses its next statement to support both sequential and concurrent operations. By following a group of assignment statements with a "next" statement, sequential execution of concurrent statement groups can be achieved.

Another special CDL statement not provided by ISP' is the "switch" statement. It is used to represent the manual switches of a computer's control panel used by the operator (4:4). The declaration

```
Switch,     POWER(ON,OFF)
```

models a power switch that can either be on or off to control power-up/power-down microoperations. If in this example,

```
/POWER(ON)/    R <- 0, A <- 1
```

the simulated power switch is set on, then the accompanying microstatements will be performed.

Finally, there are several structure declarations that are basically the same in both ISP' and CDL, but they possess minor differences worth identifying. First, a memory is declared in ISP' by specifying the keyword

"memory" followed by its identification and size specification. For example,

```
memory  Mem[0:4095]<7:0>;
```

describes a 4 kbyte memory identified as "Mem". An equivalent CDL memory statement would be

```
Register,   MAR(0-12)
Memory,     Mem(MAR) = Mem(0-4095,0-7)
```

Note that the CDL declaration specifies a memory address register (MAR) via subscript. This is because each memory is associated with a specific address register and it must be used to address a given location (4:4). In contrast, ISP' memories are not bound to a specific register. It enables its users to arbitrarily select any of its available registers to index into memory at any given time.

To represent register subfields, CDL uses the "subregister" statement while ISP provides a "format" statement. For example, here are statements in both CDL and ISP' to identify the subfields of a instruction register:

ISP'

```
state       IR<0:15>;
format      OPCODE = IR<12:15>,
            OPER1  = IR<6:11>,
            OPER2  = IR<0:5>;
```

CDL

```
Register,      IR(0-15)
Subregister,   IR(OPCODE) = IR(12-15),
               IR(OPER1)  = IR(6-11),
```

```
IR(OPER2)  = IR(0-5)
```

The major difference is that CDL does not allow independent naming of its subregisters. The parent register must precede the subscripted subregister name (4:3).

One final item worth mentioning is the difference between ISP' and CDL bus structures. A bus declaration in CDL such as:

```
Bus,  IABUS(0-31),
      EXDBUS(0-15)
```

specifies a 32-bit internal address bus and a 16-bit external data bus. However, this declaration differs from the "register" statement only by their keywords. The declaration

```
Register,  IABUS(0-31),
           EXDBUS(0-15)
```

would provide functionally equivalent model components. The capabilities of the "register" and "bus" statements are the same; each provides a storage element capable of being modified by an assignment operation.

ISP' has no special statement to model internal data buses. Therefore, as in the example above, registers would be used in their place. But ISP' does have a declaration to represent an external data bus; the "port" statement. The "port" statement does have special significance, however. Example:

```
port     EXDBUS<0:15>;
```

declares a 16-bit external data bus named EXDBUS. The port EXDBUS provides a means of communicating with external ISP' processes that comprise the system being modeled. Ports are "connected" to the ports of other ISP' components to model a system's communication and control links.

Effects Of Language Differences

Having described the basic inherent differences that exist between ISP' and CDL, it now becomes necessary to point out the effects of these differences on the CDL-to-ISP' model transformation process. As mentioned in Chapter II, the foremost objective of the transformation process is to produce an ISP' model that is functionally equivalent to its CDL counterpart. An additional self-imposed constraint was to attempt to create as much one-to-one correspondence between CDL and ISP' statements as possible to make model equivalency more readily apparent and aid in the model development and debugging process.

Similarities in the declaration statements of both CDL and ISP' made transformation of hardware component statements relatively easy and straightforward. However, the transformation of the microstatements was not as direct: the principal reason being the significant differences that exist in the way timing and control information are represented. In this section, differences in the model declaration sections, as well as those major differences in

the representation of timing and control information that had a broad impact on the way in which all instructions were transformed are identified.

Registers. The declaration sections from both the MC68000 CDL model and the resulting ISP' equivalent are presented in Appendix B. The first minor change made during the transformation was that the address and data registers were declared as an array of registers in the ISP' model rather than by individual statements. Both have equivalent results; the ISP' declaration only reduced the number of register statements (CDL has an equivalent capability but was not used). One additional difference is that CDL uses the convention that the most significant bit is determined by the size of its numerical bit designators. In ISP' the most significant bit always occupies the leftmost position regardless of its numerical designator. For example,

CDL	ISP'
PC(0-31),	PC<0:31>,

in the above declarations the most significant bit of the CDL register declaration would be PC(31), while it would be PC<0> for the ISP' declaration. To compensate for these differing conventions, ISP' registers are declared with its register bounds reversed so that microstatements referencing register bit positions would not have to be changed (i.e., PC<31:0>). In this way, a reference to bit 31 of PC would specify the most significant bit in either language model.

Subregisters. The next major difference occurs in the subregister statements. Since ISP' format statements allow the independent naming of register subfields, the parentheses enclosing subregister subscripts have not been included in its subregister declarations. For example, the CDL declaration

```
PC(LOW) = PC(0-15),
```

becomes

```
PCLOW = PC<15:0>,
```

in ISP'.

Buses. As mentioned earlier, the ISP' port is provided to allow communications with other external components of the system being modeled. The CDL external bus declarations (DBUS and ABUS) do not have a special simulation function, but only serve to represent a bus structure. Even though these external buses are declared as ports in the ISP' description, they are unconnected and nonfunctional since there are no external system components. And since there are no ISP' structures representing internal buses, equivalent register statements are used to describe internal buses in the manner of the earlier example (page V-10).

Decoders. Hamby and Guillory used CDL decoder statements to decode the instruction register to provide the control information to trigger the microstatements necessary to execute the instruction (8:VI-8). They are:

```
Decoders,   A(0-3)=IR(14-15),
            B(0-3)=IR(12-13),
            C(0-7)=IR(9-11),
            D(0-7)=IR(6-8),
            E(0-7)=IR(3-5),
            F(0-7)=IR(0-2),
            G(0-15)=IR(8-11),
            H(0-3)=IR(6-7),
```

The Motorola's MOVE.W D1,D2 instruction can be used to demonstrate their use. Since this instruction's binary representation is 0011010000000001, the instruction register's portion of the label's control expression becomes

/A(0)*B(3)*C(2)*D(0)*E(0)*F(1)*..../ .

Since decoder structures do not exist in ISP, those above were eliminated from the ISP' model. Instead, the instruction register decoding process was described by an ISP' "case" statement that used the instruction register as the evaluated expression. As an example, the ISP' case statement

```
case IR
     0b0011010000000001: MOVEWD1D2
     0b0100111011010000: JMPA0
esac
```

would execute the procedure MOVEWD1D2 or JMPA0 depending on the binary value in the instruction register IR. The procedures MOVEWD1D2 and JMPA0 would contain the microstatements necessary to accomplish their respective instructions. This example is the initial instance where the differences in the handling of control information in

nonprocedural CDL requires significant changes during the development of its ISP' equivalent.

Clock. ISP' does not support CDL's explicit two-phase clock declaration used in Hamby and Guillory's 68000 model. Because of its clock capability, CDL is best suited for synchronous systems while ISP' is oriented towards asynchronous systems (2:148). Timing signals necessary for processor operation in ISP' are normally provided by an externally-modeled clock forming part of a system. The absence of this independent, specialized timing component was also a major cause of model differences.

There were basically two approaches that could be used to circumvent this difference. First, an external clock could be modeled that provided alternating phase-one and phase-two signals. These signals would be received via ports to support the activation of "wait" statements. As an example, listed below are several execution statements extracted from the CDL model of one of the 68000's MOVE instructions.

```
/ctrl*K(2)*P(1)/  IF (DTACKN=0) THEN
                     (T<-CountupT)
/ctrl*K(2)*P(2)/  T<-CountupT
/ctrl*K(3)*P(1)/  IF (DTACKN=1) THEN
                     (T<-CountdnT)
/ctrl*K(3)*P(2)/  EXDBUF<-DBUS,
                  T<-CountupT
```

Each label's control expression contains three elements of timing and control. "Ctrl" represents the decoded contents of the instruction register used to select the execution

statements for a particular instruction. The timing and sequencing of the execution statements for that instruction are governed by the "K" and "P" elements. The K elements are timing signals from a clock-cycle counter. These signals are created by declaring a 8 X 256 decoder per the earlier CDL example:

Decoder, K(0-256)=T(0-7) .

The binary value of T will translate into one and only one of K's 256 output signals that can be used to count clock cycles and sequence through CDL execution statements.

Decrementing the value of the control register T via the "CountdnT" operator also changes the corresponding state of K's control signal output. In this way, one is able to recover a clock cycle. The above CDL routine represents a loop that allows the processor to wait for DTACKN to become low. CDL's ability to segregate control and data, while at the same time enabling control to be modified via its microoperations, is a major stumbling block in the efficient transformation from CDL to ISP'. With ISP', the control provided by K must be handled by conditional statements that become part of the ISP' microstatements themselves. Efficient transformation is also exacerbated by the fact that alternating phase-one and phase-two timing signals are automatically provided by an external clock to provide an additional level of sequencing within each clock cycle. Assuming an externally modified two-phase clock, the ISP'

model that follows is equivalent to the previous CDL routine.

```
case ctrl
     decoded instruction 1: 1st instruction's routine
     decoded instruction 2: 2nd instruction's routine
                  .         :            .
                  .         :            .
                  .         :            .
     decoded instruction n: last instruction's routine
esac;

instruction n's routine :=
                          (
                          .
                          .
                          .
                          while K<2>
                                (
                                wait(phi1:lead);
                                if not DTACKN
                                   (
                                   T = T + 1;
                                   delay(1)
                                   );
                          while K<2>
                                do
                                  (
                                  wait(phi2:lead);
                                  T = T + 1;
                                  delay(1);
                                  while K<3>
                                        (
                                        wait(phi1:lead);
                                        if DTACKN
                                           (
                                           T = T - 1;
                                           delay(1)
                                           )
                                        )
                                  )
                                until not DTACKN;
                           while K<3>
                                 (
                                 wait(phi2:lead);
                                 EXDBUF = DBUS;
                                 T = T + 1;
                                 delay(1)
                                 );
                          .
                          .
```

Clock and Clock-Cycle Counter Representation. From the above CDL and ISP' examples, a couple of facts emanating from the differences between procedural ISP' and nonprocedural CDL become clear:

1) the models become dissimilar in appearance, and the equivalency of the models becomes obscured by differences in the method in which timing and control signals are represented.

2) the flow of execution of the ISP' model is more difficult to follow because of the infusion of the conditional constructs (i.e., while, wait, do...until) into the microstatements in order to represent the timing and control.

These elements have an adverse impact on the process of establishing model equivalency that is essential to validation of the models developed by Hamby and Guillory. Such diversity would also hamper attempts to locate causes of differences between the simulation results and the documented logic analyzer output.

One fact not made clear by the above examples is that the ISP' simulation will be much more complex because of the additional modeling of the clock-cycle counter and two-phase clock components. Modeling these devices, and the synchronization that must be achieved between the resulting system components, significantly increases the complexity of the simulation's construction.

An alternate ISP' representation of the above sequence

that includes an internal two-phase clock and control register is depicted below.

```
        .
        .
        .
T = 2; next
phi1 = hi;
phi2 = lo;
while DTACKN eql hi
        (
        next;
        phi1 = lo;
        phi2 = hi; next
        T = 3; next
        phi1 = hi;
        phi2 = lo; next
        T = 2
        );
next;
T = 3; next
phi1 = lo;
phi2 = hi;
EXDBUF = DBUS; next
        .
        .
        .
```

One first notes that the clock-cycle counter K is not present. Although it better depicts the actual operations taking place within the microprocessor, it has been removed because it is nonessential to the model's accuracy. The condition K<2> is equivalent to the condition T eql 2. Thus the elimination of decoder K resulted in a simpler model by removing an unnecessary level of indirection.

The wait and while statements associated with the control signals have also been eliminated. This was achieved by modeling an internal two-phase clock and clock-cycle counter. The processor no longer relies on

external components to provide these signals. It provides them itself by including statements that represent the required clock states at appropriate points in an instruction's microstatement sequence. Thus, rather testing for K<3> and then waiting for phil, it directly sets T = 3 and phil = hi and then executes the microstatements associated with phase one of clock cycle three.

By internally modeling these components, the speed of the simulation increases because the process representing the microprocessor is not idled by the delay and wait statements. These statements transfer simulation execution to the other system components by placing the 68000 model in the "wait" state and then "running" the process representing either the external clock or clock-cycle counter as appropriate (6:3). This continual process-swapping severely slows the simulation. Also, because the frequency of the 68000's host clock was not reflected in the original CDL model, the delay statements become unnecessary. The models are functionally equivalent, only the later does not reflect independent timing and control signals. The ISP' example also assumes a correctly functioning clock and control register, which is acceptable when not modeling hardware failure.

This representation brings the CDL and ISP' models more in-line with one another. In the ISP' model, separate timing and control signal statements now lead their corresponding microinstructions. This representation makes

equivalency easier to establish and greatly simplifies development of the simulation system. An external clock and clock-cycle counter no longer has to be modeled and interfaced with the MC68000. For these reasons, the second ISP' modeling approach was selected. Rather than strive for the segregation of control and timing information from the microstatements as is done in the real world and handled so nicely by CDL, these elements are internally modeled to enhance model semblance and reduce simulation complexity, while at the same time preserving the functional accuracy of the MC68000 model.

Switch. Because ISP' does not support a "switch" statement, it was modeled with a single bit register named "SWITCH" that is either set to high to simulate system power on, or low to simulate power off.

Model Changes

Now, modifications made to the model that are not the result of the inherent differences that exist between CDL and ISP' are presented. These changes or additions were made to enable a functioning simulation, initialize the model's storage components, or to modify component declarations. Changes peculiar to a particular instruction are discussed in the next chapter which analyzes the results of the model's simulation.

Memory Responses. Upon reexamining the earlier CDL example,

```
/ctrl*K(2)*P(1)/  IF (DTACKN=0) THEN
                     (T<-CountupT)
/ctrl*K(2)*P(2)/  T<-CountupT
/ctrl*K(3)*P(1)/  IF (DTACKN=1) THEN
                     (T<-CountdnT)
/ctrl*K(3)*P(2)/  EXDBUF<-DBUS,
                  T<-CountupT
```

one notices that during the simulation the value of DTACKN must be low for these execution statements to complete, else the routine would be caught in an infinite loop executing the second and third CDL execution statements. Because Hamby and Guillory did not intend to externally model the ECB's memory, the activities associated with that memory's processing of the data and control signals that exist between it and the 68000 (i.e., R_W, ASN, LDSN, UDSN, ABUS, DBUS) do not appear in their CDL model of the 68000. However, in order to produce a functioning simulation, the memory's responses to actions initiated by the 68000 must be represented. Again, one becomes faced with the decision to either externally model the ECB's memory to represent the real-word environment, or as was done with the timing and control signals, place the memory responses in-line with the 68000's microinstructions at points that coincide with the results observed by Hamby and Guillory on the logic analyzer and documented in Chapter VI of their thesis.

For example, according to the logic analyzer output for the MOVE.W D1,D2 instruction, the ECB's memory placed the instruction requested during its prefetch cycle on the data bus during phase one of clock cycle three. It also took DTACKN low at that time indicating that valid data was on

the bus. Although the CDL model does not reflect the memory's responses, they must be modeled somewhere to provide a working simulation. To simplify the construction of the simulation, memory responses were inserted in-line with the code in accordance with the logic analyzer results. This does not detract from the model's accuracy. Accordingly, the ISP' model becomes

```
        .
        .
        .
T = 2; next
phi1 = hi;
phi2 = lo;
while DTACKN eql hi
        (
        next;
        phi1 = lo;
        phi2 = hi; next
        T = 3; next
           phi1 = hi;
           phi2 = lo;
           DBUS<15:8> = M[ABUS];
           DBUS<7:0> = M[ABUS + 1];
           DTACKN = lo; next
        T = 2
        );
next;
T = 3; next
phi1 = lo;
phi2 = hi;
EXDBUF = DBUS; next
        .
        .
        .
```

Now the memory location's contents specified by the address bus (ABUS) is placed on the data bus (DBUS) and DTACKN is taken low. This will enable the MOVE instruction to prefetch the next instruction and proceed with its execution. Memory reads or writes requiring wait states

were modeled by including a "wait-cycle" counter in the loop to accomplish the data transfer on the correct clock cycle.

EXABUF Size. Hamby and Guillory declared a 16-bit external address buffer with the "EXABUF(0-15)" statement. Because the MC68000 has a 23-bit address bus, the buffer size was enlarged to 23 bits with the following ISP' port declaration:

```
EXABUF<23:1>, .
```

ABUS Utilization. In none of the instruction sequences did Hamby and Guillory explicitly move the contents of the external address buffer (EXABUF) to the address bus (ABUS). In practice, when the data is loaded into EXABUF, its contents would automatically appear on the address bus (if ABUS had been enabled). Subsequently, there is an implied transfer of an address to the address bus when EXABUF is loaded. However, since the CDL bus declaration does not provide any physical relationship between any of its declared buses and system registers, such a transfer must be accomplished explicitly by the statement "ABUS = EXABUF". This additional address transfer allowed a more accurate representation of the memory addressing mechanism. Now, rather than identifying a memory location with the ISP' statement

```
DBUS<15:8> = M[EXABUF];
```

the statement

```
DBUS<15:8> = M[ABUS];
```

can be used and more accurately describes the memory addressing process.

Memory Declaration. Hamby and Guillory also specified an eight-megaword memory with the declaration "Memory M( ) = M(0-8388607,0-15)" (8:VI-8). This corresponds to the MC68000's maximum physical addressing range. However, to more accurately represent the ECB environment, the memory size was reduced to 32 kbytes with the declaration "memory M[0:32767]<7:0>" without affecting the model's functional accuracy.

High Impedance Representation. The entire data and address buses were placed in a high state to simulate their high impedance states whenever not being used by the 68000 or its memory. Although not modeled by Hamby and Guillory, this matches the logic analyzer's output. The ISP' statements

```
DBUS = 0xffff;
ABUS = 0xffffff;
```

were added to the ISP' microstatements as needed to place the data or address bus in a high impedance state whenever appropriate (i.e., DBENABLE, ABENABLE = lo). Likewise, the ECB's empty memory locations are also in a high state. Therefore, simulated memory locations immediately following the JMP (A0) instruction of all test routines are initialized to the high state to meet the observed results

(unused memory locations are set to a low state by N.mPc).

Power-On Sequence. A power-on and initialization routine (power_on_initialize) was added to the model to:

1) accomplish the 68000's power-on sequence as described by Hamby and Guillory (8:VI-9),

2) initialize appropriate registers prior to each test routine as specified by Hamby and Guillory in Appendix A of their thesis,

3) initialize the data and address buses to the high impedance state,

4) initialize the 68000's active low memory control signals to the high state, and

5) initialize memory to the high state.

Additional Simulation Components

Not relevant to the MC68000 microprocessor model's accuracy, but important to the development of the simulation package, is the development of Metamicro and Linking/Loader programs. In addition to the model of the 68000 processor, a description of the 68000's instruction set had to be developed to transform the MC68000 assembly language test routines into executable code for the processor model. A loader is needed to initialize the 68000's internal memory with this code. Because N.mPc's library included programs that could be modified to accomplish these functions, they were used rather than undertake an extensive and redundant development effort. The Metamicro and Linking/Loader descriptions used were developed by Samir Shah while a

graduate student at Case Western Reserve University and are included in appendices D and E respectively with the modifications described below.

Program Loader. One of the changes made to Shah's Linking/Loader description was an alteration of the "space " statement. To reflect the fact that all instruction test routines were loaded into the ECB's 32-kbyte memory beginning at address location 1000 hex, the space declaration for them was changed from "space<0:4095>$" to "space <4096:32767>$". Because the exception processing routines required use of lower memory to support the vector table and system stack area, their Linking/Loader description's space declaration became "space <0:32767>$". Program location was then accomplished within Metamicro. To support the use of labels within a program, the "mode" declaration was enhanced with the following statements:

```
case Opcode eql 6:
     I1 = address - . - 2 $
     break$
esac,
```

And finally, the assignment "I0 = 0x4d $" within the "transfer" statement was changed to "I0 = 0x4e $" to accurately represent the JMP instruction's format.

MC68000 Assembler. Hamby and Guillory initialized bit 13 of the status register (mode selection bit) to zero with the priviledged instruction "AND.W #$DFFF,SR" to indicate the user mode. Because Shah did not model priviledged

instructions in his Metamicro description of the 68000, this instruction was not included in the early simulation test routines. Instead bit 13 of the status register was set to zero in the power-on and initialization routine without model degradation because the 68000's output signals were only examined when the instruction of interest was executing and not during the execution of the AND.W instruction. Since this instruction was omitted from the test routines, address register A[0] was loaded with 1000 hex instead of 1004 hex to enable the JMP (A0) to function correctly. Once Metamicro was mastered, Shah's Metamicro description of the 68000's instruction set was modified so it could assemble the "AND.W #$1000,SR" instruction. As a result, some later routines include this instruction. In those that do, the PC and A[0] are initialized to 1004 hex in the instruction's ISP' description. These routines also use the later version of the modified Linking/Loader program.

From the test routines in Appendix D, one notes that basic differences exist between them and the standard MC68000 assembly code. They are:

1) the operand's size specification is segregated from the instruction mnemonic,

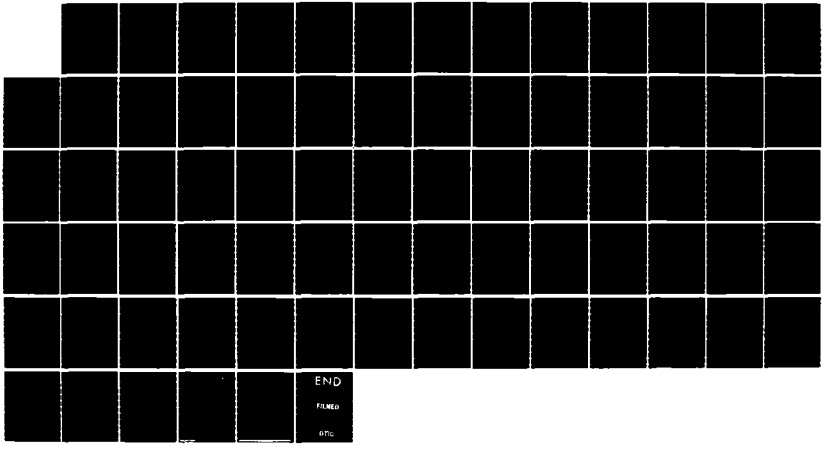
2) prefixes are incuded to specify each operand's addressing mode (special symbols such as "#" and "$" are not recognized by Metamicro),

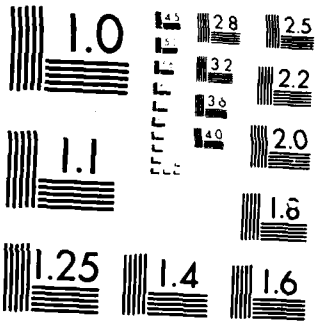
3) all operands are separated by commas, and

4) parenthesis surround an instruction's operands.

AD-A164 203 THE SIMULATION AND ANALYSIS OF A RTL MODEL OF THE MOTOROLA MC68000 MICROP..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. C A BAXLEY 2/2

UNCLASSIFIED DEC 84 AFIT/GCS/ENG/84D-2-VOL-1 F/G 9/2 NL





These changes are mandated by Metamicro macro-oriented structure.

VI. Simulation Analysis

Introduction

An ISP' model was constructed and then simulated for each CDL instruction or exception model developed by Hamby and Guillory. In addition to the MOVE (16 variations), JMP, ADD, BEQ, and BTST instructions, they also included the Illegal Instruction and Address Error exceptions. The results of each simulation were carefully compared with the logic analyzer data tabulated by Hamby and Guillory in Chapter VI of their thesis. Differences were analyzed for their causes and they in turn were evaluated for their severity. A difference's impact on the validity of this approach to microprocessor modeling could be gauged in terms of the answer to the following question: could the noted difference be explained?

Regardless of their numbers and type, abnormalities whose causes could be isolated, could also be eliminated by correcting either the model or logic analyzer output data as appropriate. Although it is possible to model unexplained (but predictable) behavior, many differences (especially those that are repetitive) that cannot be satisfactorily interpreted would strongly suggest that this particular approach is not practical when a highly accurate model is the objective.

Because there were no differences that were unexplanable, the results of the simulations' analyses

indicate that the models developed by Hamby and Guillory accurately describe the MC68000's behavior when processing the above instructions and exceptions. Differences that were considered major (primarily because of their repetitiveness), centered around the state of the address and data buses (ABUS and DBUS) during transitional periods and had little effect on the functional accuracy of the model. The consistent accuracy of Hamby and Guillory's models strongly supports this approach as a microprocessor modeling technique.

In this chapter the overall results of the analyses are first presented. Here the results of the individual model analyses have been consolidated and evaluated to identify the major or consistent differences that reflect on the viability of this modeling approach. These differences are identified, interpreted, and then individually and collectively assessed for their impact on this approach. This discussion is followed by sections that detail the simulation results for the individually modeled instructions or exceptions from which the assessment of this approach was formed.

Analysis Results

Based on the analysis of the simulation results, there is little doubt that microprocessors can be accurately modeled through signal analysis with minimal supporting technical data. Except on those rare occasions when microstatements were incorrect or omitted, the simulation

results deviated little from the analyzer output, particularly with the 68000's memory control signals (i.e., DTACKN, AS, UDS, LDS, and R/W). However, as mentioned there were some deviations from the data and address bus states reported by the logic analyzer whenever these buses would transition to the high impedance state at the beginning of a read or write cycle. The differences were consistent and predictable throughout the instruction models and result from the 68000's inability to complete its activities well within the logic analyzer's sampling intervals. These bus differences do not have a major impact on the accuracy of the models because they are of short duration (half clock cycle) and they occur at points where the ABUS or DBUS are not being monitored by either the processor or its memory.

The problem of accurately modeling the high impedance state is really inconsequential to the functional correctness of the model; however, a major section is devoted to discussing this problem, even at the risk of overstating its significance. With so few global inconsistencies in the models, considerable attention can be given to this anomaly. The accuracy and completeness of the models developed by Hamby and Guillory is a tribute to their hard work and thorough understanding of microprocessor structure and operation.

High Impedance State

The differing times in which the 68000 and its supporting memory release the data and address buses during

read and write cycles made their state difficult to accurately model during the transition to a new cycle. The specific problem areas are addressed in detail in the sections that follow.

Data Bus. The available technical data specifies that during a read cycle the memory must remove its data and data transfer acknowledge (DTACK) signal within one clock period of recognizing the 68000's negation of the address strobe (AS) that occurs during phase two of the last read clock cycle (11:38). On the other hand, during a write cycle the technical data (1:4) specifies that the 68000 will release the data bus no earlier than 60 ns from AS negation (no maximum figure is provided). AS is also negated during phase two of a write's last clock cycle. Based on the logic analyzer results, it appears that the ECB's memory releases the data bus much sooner after the completion of a read cycle than the 68000 does when completing a write cycle.

For a read or write cycle, the logic analyzer finds the data bus in the high impedance state during phase one of their initial clock cycle - provided they do not follow another write cycle. Should they happen to follow another write cycle, then the logic analyzer does not see the data bus returned to the high state until phase two of their initial cycle. While the specific point in which the bus is "high-impedanced" cannot be determined for either the read or write cycle, time intervals for each can be estimated using available timing information.

The logic analyzer samples data at the low and high transitions of the system clock in accordance with Figure VI-1. The figure shows that for any clock period, the phase one sample will reflect changes that occur with the clock signal high and phase two will catch state changes that occur during the low clock signal.

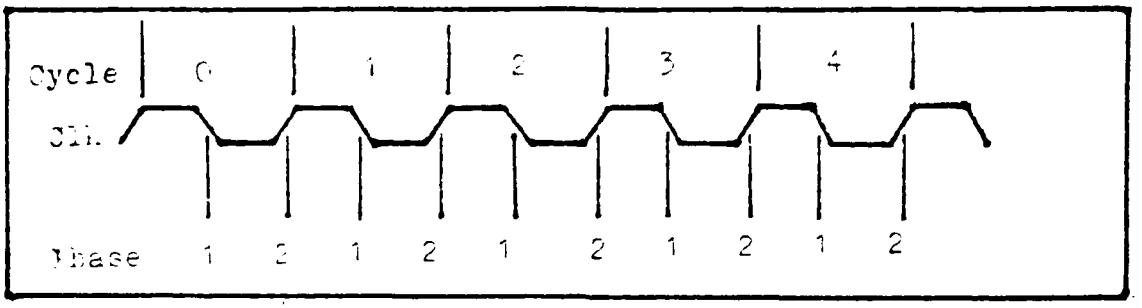


Figure VI-1. Logic Analyzer Sampling Points

Performance specifications for the Hewlett Packard 1600A Logic State Analyzer indicate that data must be present at least 20 ns prior to clock transition for it to be captured during the current phase (24:1-1). Figure VI-2(a) presents the maximum time (120 ns) in which the 68000 will place the data bus in the high impedance state upon entering a read or write cycle (1:4). In order for the logic analyzer to capture the high impedance state during phase one, then the bus must be in this state within 95 ns of the beginning of the read or write cycle. It appears that the 68000 disables the bus somewhere between 86 and 110 ns after the start of the first clock cycle. Because the signals have a 10 ns rise/fall time (1:4), the bus will reach the high impedance state in 96 to 120 ns (Figure VI-2(b)). These figures indicate that the logic analyzer

should not show the data bus returning to the high impedance state until phase two of the first clock cycle for both read and write cycles. This is not the case.

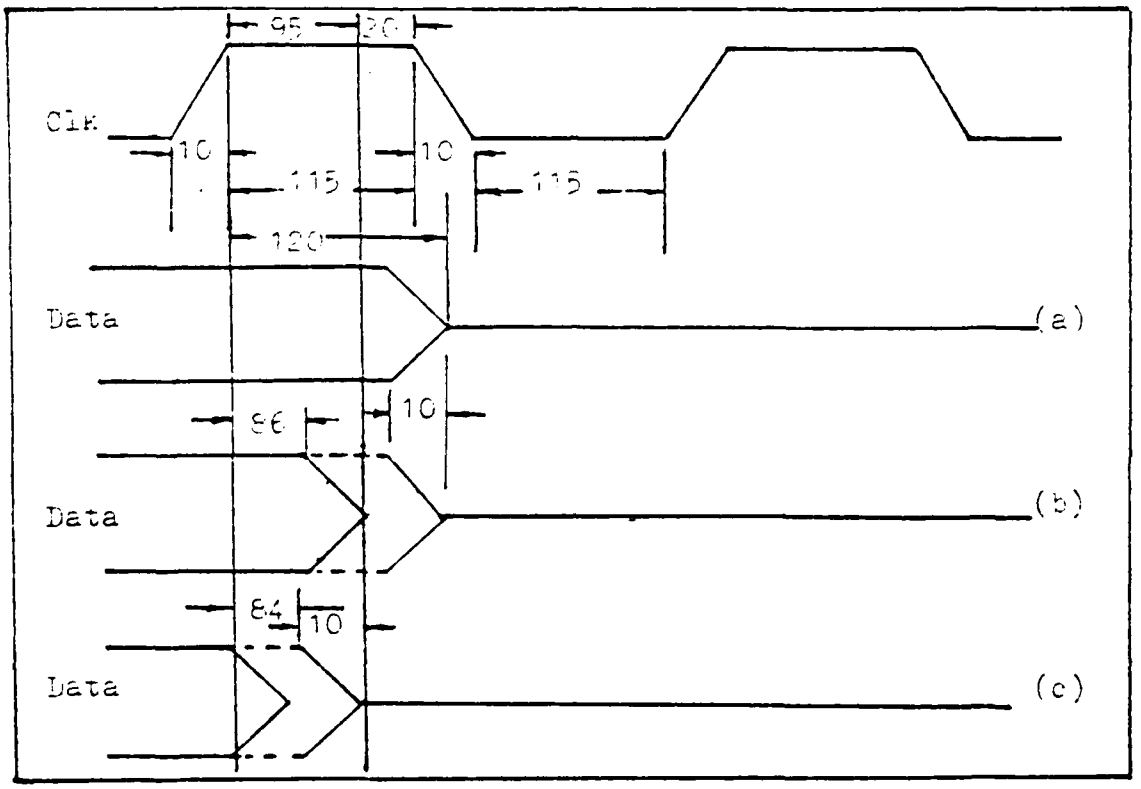


Figure VI-2. Data Bus High Impedance Timing (1:5,6)

Since the 68000 does not place the data bus in a high impedance state in sufficient time to be captured by the logic analyzer on phase one of clock cycle zero for either a read or write cycle, then the only explanation for the logic analyzer results is that a previous cycle is sometimes able to return DBUS to high impedance so that it is captured by the logic analyzer during phase one, even though the current cycle fails to do so itself.

During a read cycle it is the memory's responsibility to return the data bus to the high impedance state once the

68000 has received the requested data. As mentioned earlier, it has up to a full clock cycle from phase two of the last read clock cycle to do this. The change could take place as late as phase two of the following clock cycle. However, it appears that the ECB memory responds much faster than necessary. Referring to Figure VI-2(c), the ECB memory must disable DBUS no later than 85 ns into the following clock cycle because it is being captured by the logic analyzer during phase one. In the case where the 68000 is responsible for returning the bus to the high impedance state (end of a write cycle), it fails to do so in time to compensate for the initial clock cycle's failure. In fact, it is possible that the 68000 relies on the following read or write cycle to accomplish this.

Hamby and Guillory's model specifies that the data bus is disabled during phase one of both the read and write clock cycles and is reflected in the initial simulation results. However, because the 68000 does not accomplish this action in time during a write cycle to be captured by the logic analyzer until the next phase, it appears as though the model is inaccurate. The technical data supports their model even though the timing resolution of the logic analyzer disguises its accuracy.

In terms of accurately modeling any instruction, one must then be aware of the types of cycles and their ordering so that the state of the data bus can be accurately portrayed at any given time. That is, one must know whether

the previous cycle was a read or write so that the correct variation of the current cycle can be determined and modeled to place the data bus in the high impedance state at the correct moment. This adjustment was made during the model transformations once the problem was recognized. It is important to note that even though this was easily accomplished, it is not necessary in terms of a correctly functioning model. The state of the data bus is of no consequence during the initial phase of a read or write clock cycle because no device is monitoring it at that point. During either a read or write cycle, the 68000 can receive or issue valid data no sooner than a full clock cycle later.

Address Bus. A similar situation exists whenever the address bus (ABUS) is transitioning to the high impedance state. As with the data bus, the 68000 places ABUS in the high impedance state no later than 120 ns from the beginning of a read or write cycle. The logic analyzer results indicate that ABUS is in an indeterminate state during phase one of the initial period in a read or write cycle. In some instances the address bus was found in a high impedance state during phase one; however, there were just as many other cases where it contained the address from the previous read or write cycle, or it was in a state somewhere between the two (previous address and high impedance). This strongly indicates that the logic analyzer was sampling the data at approximately the same instant in which the 68000

was returning the bus to the high impedance state.

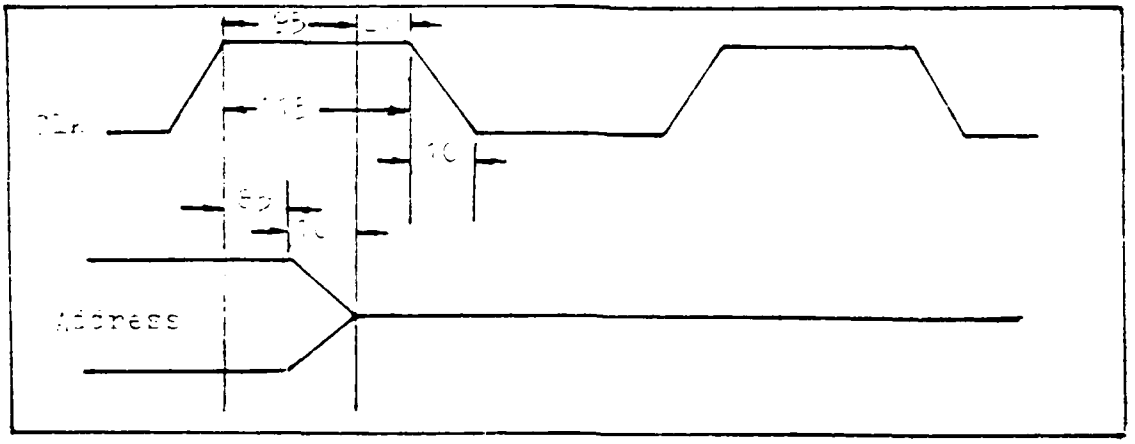


Figure VI-3. High Impedance On Address Bus

Looking at Figure VI-3, it appears that this is occurring at approximately 85 ns into the initial clock cycle. If the 68000 disables the bus at 85 ns into its cycle, then with a 10 ns rise/fall time the bus will return to high impedance at 95 ns (the cutoff time for the logic analyzer to capture it). Timing is critical and a delay of as little as a single nanosecond will mean the difference between the logic analyzer catching the address bus in the high impedance state during phase one or not.

As with data bus modeling, even though the simulation results will not always agree with the results of the logic analyzer (the simulation will always reach the high impedance state), this indeterminate state lasts but a half clock cycle and occurs at a point where the bus is of no consequence to system devices. For both read and write cycles, the 68000 does not take AS low to indicate a valid address until a full clock cycle later.

Individual Instruction Simulation Results

The simulation results for each modeled instruction will be individually compared and contrasted with logic analyzer's reflection of the 68000's actual operation. Each section contains a brief description of the instruction or exception, lists the simulation data, and then explains their differences and corrections. The individual ISP' models of each instruction or exception sequence, the files that controlled the simulations, and the simulation output from which the tabulated signal data was prepared appear in Appendices C, G, and H respectively. Appendix F lists the instruction test routines that were executed on the models during the simulations. The CDL models and logic analyzer data from which the comparisons were made are contained in Appendices I and J respectively. All addresses and data values used in the discussion are given in hexadecimal.

Excluding wait cycles, read cycles require four clock cycles to execute. The ECB memory required one and a half clock cycles to provide the requested data causing the 68000 to run a full wait cycle during its reads. Subsequently, all read cycles require five clock cycles to complete. During writes to memory, the 68000 must also wait one and a half cycles before the ECB memory formally acknowledges data receipt. With the wait cycle included, a normal four-cycle write will also require five clock cycles to execute. As a matter of convention, all instruction cycle lengths given in this chapter will include wait cycles.

When running the simulations, 16 of the MC68000's 64 pins were monitored to provide sufficient information to identify the internal processing taking place. These signals along with the columns in which their states appear in the simulation data are identified below.

Column	Signal
0	FC0 (Function Code 0 output)
1	FC1 (Function Code 1 output)
2	FC2 (Function Code 2 output)
3	DTACK' (Data Transfer Acknowledge input from memory)
4	R/W' (Read/Write signal)
5	LDS' (Lower Data Strobe)
6	UDS' (Upper Data Strobe)
7	AS' (Address Strobe)
8	D0 (Data line 0)
9	D1 (Data line 1)
10	D2 (Data line 2)
11	D3 (Data line 3)
12	D4 (Data line 4)
13	D5 (Data line 5)
14	D6 (Data line 6)
15	D7 (Data line 7)

For several of the instructions, it was necessary to monitor the address bus as well. Whenever this was done, only the first four data lines (D0-D3) were monitored. D4-D7 were replaced with A1-A4 and the results entered in columns 12-15 of the simulation data. The only exception to this convention occurs while simulating the Illegal Instruction exception sequence. In this lone case, address lines A1-A4 were monitored in addition to the eight data lines D0-D7 to provide greater data range when also observing addresses. For this instruction, A1-A4 appear in columns 16-19.

MOVE.W D1,D2

The MOVE.W D1,D2 instruction was the first to be modeled and simulated. This single-word instruction uses the data register direct addressing mode to move the contents of data register D1 to data register D2. Because this instruction is executed in a single read cycle (prefetch), it required only five clock cycles to execute. Only the function code, peripheral control, and data lines were monitored during the simulation. The simulation results were:

```
                15-12 11-8 7--4 3--0 (columns)
                 DDDD DDDD AULR DFFF (signals)
                 7654 3210 SDD/ TCCC
                           SSW A210
                               C
                               K
                ---------------------
K(0) P(1)        1111 1111 1111 1010 $Data bus (8-15) in high
K(0) P(2)        1111 1111 1111 1010 $impedance state, R/W'
                                     $signal (4) indicates a
                                     $read cycle, function codes
                                     $(0-2) identify user
                                     $program mode.
K(1) P(1)        1111 1111 0001 1010 $AS' (7) indicates a
K(1) P(2)        1111 1111 0001 1010 $valid address on address
                                     $bus; UDS', LDS' (5-6) for
                                     $a word size operation.
K(2) P(1)        1111 1111 0001 1010 $DTACK' (3) not asserted
K(2) P(2)        1111 1111 0001 1010 $by peripheral device (data
                                     $not ready) so processor
                                     $runs a wait cycle.
K(3) P(1)        0000 0001 0001 0010 $Data applied to data bus
K(3) P(2)        0000 0001 0001 0010 $(8-15), DTACK' (3) asserted
                                     $indicating to processor
                                     $that data is on the bus.
                                     $The data on the bus is
                                     $code for MOVE.W D1,D2
                                     $instruction indicating
                                     $that this is a prefetch.
K(4) P(1)        0000 0001 0001 0010
K(4) P(2)        0000 0001 1111 1010 $AS', UDS', LDS' (5-7)
```

```
$change to notify
$peripheral device that
$transfer is complete.
```

Because the simulation results echoed the results observed through the logic analyzer during the actual operation of the MC68000, Hamby and Guillory appeared to have little difficulty modeling this version of the MOVE instruction. There were, however, two changes made to their model during its transformation to ISP'. At phase two of clock cycles zero and three, decoder K was directly incremented with the CDL statement "K <- CountupK". Incrementing K directly would not achieve the desired results. The decoder's successive output line can be activated by incrementing its input clock cycle register, T, with the CDL statement "T <- CountupT" (T = T + 1 in ISP').

MOVE.W D1,(A1)

The MOVE.W D1,(A1) instruction uses both the data register direct and address register indirect addressing modes to move the contents of data register D1 into the memory locations identified by the contents of address register A1. This version of the MOVE instruction consisted of single read and write cycles requiring 10 clock cycles to execute. During the simulation, no address lines were monitored and the results did not differ from the logic analyzer data. They were:

```
                  15-12 11-8 7--4 3--0 (columns)
                   DDDD DDDD AULR DFFF (signals)
                   7654 3210 SDD/ TCCC
                             SSW A210
                                 C
                                 K
                  ---------------------
K(0) P(1)          1111 1111 1111 1010 $Data bus (8-15) in high
K(0) P(2)          1111 1111 1111 1010 $impedance state, R/W'
                                       $signal (4) indicates a
                                       $read cycle, function codes
                                       $(0-2) identify user
                                       $program mode.
K(1) P(1)          1111 1111 0001 1010 $AS' (7) indicates a
K(1) P(2)          1111 1111 0001 1010 $valid address on address
                                       $bus; UDS', LDS' (5-6) for
                                       $a word size operation.
K(2) P(1)          1111 1111 0001 1010 $DTACK' (3) not asserted
K(2) P(2)          1111 1111 0001 1010 $by peripheral device (data
                                       $not ready) so processor
                                       $runs a wait cycle.
K(3) P(1)          1000 0001 0001 0010 $Data applied to data bus
K(3) P(2)          1000 0001 0001 0010 $(8-15), DTACK' (3) asserted
                                       $indicating to processor
                                       $that data is on the bus.
                                       $The data on the bus is
                                       $code for MOVE.W D1,(A1)
                                       $instruction indicating
                                       $that this is a prefetch.
K(4) P(1)          1000 0001 0001 0010
K(4) P(2)          1000 0001 1111 1010 $AS', UDS', LDS' (5-7)
                                       $change to notify
                                       $peripheral device that
                                       $transfer is complete.
K(5) P(1)          1111 1111 1111 1010 $Begin write cycle.
                                       $Data bus (8-15) in
                                       $high impedance state.
K(5) P(2)          1111 1111 1111 1001 $Function code (0-2)
                                       $is user data mode.
K(6) P(1)          1111 1111 0110 1001 $AS' (7) asserted to
                                       $indicate valid address on
                                       $ bus, R/W' (4) changes
                                       $to write cycle.
K(6) P(2)          0101 0101 0110 1001 $Data put on data bus
                                       $(8-15). Data is 55 hex
                                       $which is same as data
                                       $stored in D1.
K(7) P(1)          0101 0101 0000 1001 $UDS', LDS' (5-6) identify
K(7) P(2)          0101 0101 0000 1001 $word size, DTACK' (3) not
                                       $asserted by peripheral
                                       $so wait cycle run by
                                       $processor.
K(8) P(1)          0101 0101 0000 1001 $DTACK' (3) not asserted
```

```
K(7) P(2)   0101 0101 0000 1001 $sc T is decremented and
                                $another wait cycle run.
K(8) P(1)   0101 0101 0000 0001 $DTACK' (3) is asserted
K(8) P(2)   0101 0101 0000 0001 $indicating peripheral has
                                $successfully stored data.
K(9) P(1)   0101 0101 0000 0001
K(9) P(2)   0101 0101 1110 1001 $AS', UDS', LDS' (5-7)
                                $change to signal
                                $peripheral that write
                                $cycle is complete.
```

MOVE.L D1,A1

The single-word instruction MOVE.L D1,A1 uses both the data register and address register direct addressing modes to move the 32-bit contents of data register D1 to address register A1. This instruction consists of a single read cycle that required five clock cycles to execute. No address lines were monitored. The results were:

```
                15-12 11-8 7--4 3--0 (columns)
                 DDDD DDDD AULR DFFF (signals)
                 7654 3210 SDD/ TCCC
                           SSW A210
                               C
                               K
                ---------------------
K(0) P(1)   1111 1111 1111 1010 $Data bus (8-15) in high
K(0) P(2)   1111 1111 1111 1010 $impedance state, R/W'
                                $signal (4) indicates a
                                $read cycle, function codes
                                $(0-2) identify user
                                $program mode.
K(1) P(1)   1111 1111 0001 1010 $AS' (7) indicates a
K(1) P(2)   1111 1111 0001 1010 $valid address on address
                                $bus; UDS', LDS' (5-6) for
                                $a word size operation.
K(2) P(1)   1111 1111 0001 1010 $DTACK' (3) not asserted
K(2) P(2)   1111 1111 0001 1010 $by peripheral device (data
                                $not ready) so processor
                                $runs a wait cycle.
K(3) P(1)   0100 0001 0001 0010 $Data applied to data bus
K(3) P(2)   0100 0001 0001 0010 $(8-15), DTACK' (3) asserted
                                $indicating to processor
                                $that data is on the bus.
                                $The data on the bus is
                                $code for MOVE.L D1,A1
```

```
                                        $instruction indicating
                                        $that this is a prefetch.
K(4) P(1)  0100 0001 0001 0010
K(4) P(2)  0100 0001 1111 1010 $AS', UDS', LDS' (5-7)
                                        $change to notify
                                        $peripheral device that
                                        $transfer is complete.
```

The only difference between the simulation results and the data from the logic analyzer occurs at phase two of clock cycle two. The logic analyzer shows the data bus changed from its high impedance state of the previous cycle to "1011 1111" on its way to the valid data state "0100 0001" at phase one of clock cycle three. Because this occurs at a point in the read cycle where the memory is placing data onto the bus, it appears that the logic analyzer has caught the data bus in transition to its valid data state. The simulation differs by maintaining the high impedance state during this phase.

MOVE.W D1,(A1)+

MOVE.W D1,(A1)+ is a single word instruction that uses the data register direct and postincrement register indirect addressing modes to move the contents of data register D1 to the memory location pointed to by the contents of address register A1. After A1 is used to address the memory location, it is incremented by two. It is comprised of single read and write cycles that require ten clock periods to execute.

Instead of monitoring eight data lines, only the first four were examined during the execution of this instruction. Hamby and Guillory replaced data lines D4-D7

with the four least significant address lines (A1-A4) to view the postincrement process. The reader should be aware that these lines do not include bit zero of the program counter; they only represent PC<4:1>. PC<0> does not appear on the address bus. Rather, it is internally encoded along with the instruction's operand length to activate the upper and lower data strobes (UDS and LDS) (11:38).

The 68000 uses PC<0> to determine which byte to read or write and then activates either UDS, LDS, or both depending on the operand size. Whenever the instruction specifies a byte operand, UDS is activated if PC<0> is zero. If it is a one, then LDS is activated. For word size operands, both UDS and LDS are activated and PC<0> must be zero to avoid an address error exception. Therefore, for the word or longword size instructions that dominate the models, an extra zero should be appended to the address bus data to arrive at the correct memory address accessed by the processor. In terms of an accurate representation of this process in the instruction models, the sequence of CDL microstatements that transfer the program counter to the address bus (i.e., IABUS <- PC, EXABUF <- IABUS, and ABUS <- EXABUF) should be changed at both of the first two stages to support this fact (i.e., IABUS<31:1> <- PC<31:1> and EXABUF <- IABUS<31:1>). The simulation results were:

```
                15-12 11-8 7--4 3--0 (columns)
                 AAAA DDDD AULR DFFF (signals)
                 4321 3210 SDD/ TCCC
                           SSW A210
                              C
                              K
                ---------------------
K(0) P(1)  1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)  0011 1111 1111 1010 $Address lines (12-15)
                               $are 1006, location of
K(1) P(1)  0011 1111 0001 1010 $instruction being
K(1) P(2)  0011 1111 0001 1010 $prefetched

K(2) P(1)  0011 1111 0001 1010
K(2) P(2)  0011 1111 0001 1010

K(3) P(1)  0011 0001 0001 0010 $Data applied to data bus
K(3) P(2)  0011 0001 0001 0010 $(8-11). Data is code
                               $for MOVE.W D1,(A1)+.

K(4) P(1)  0011 0001 0001 0010
K(4) P(2)  0011 0001 1111 1010 $End read cycle

K(5) P(1)  1111 1111 1111 1010 $Begin write cycle.
K(5) P(2)  0000 1111 1111 1001 $Address lines (12-15)
                               $are 2000.
K(6) P(1)  0000 1111 0110 1001
K(6) P(2)  0000 0101 0110 1001 $Data put on data bus
                               $(8-11). Data is 5 hex
K(7) P(1)  0000 0101 0000 1001 $data being moved.
K(7) P(2)  0000 0101 0000 1001

K(8) P(1)  0000 0101 0000 1001
K(7) P(2)  0000 0101 0000 1001

K(8) P(1)  0000 0101 0000 0001
K(8) P(2)  0000 0101 0000 0001

K(9) P(1)  0000 0101 0000 0001
K(9) P(2)  0000 0101 1110 1001 $End write cycle.
```

In their CDL model, Hamby and Guillory neglected to increment the program counter or place the contents of the prefetch register into the instruction register during phase two of clock cycle nine. These microstatements were added to the ISP' version. The only difference that occurred between the simulation and the logic analyzer data is the

data that appears on the address bus at phase one of clock cycle zero. The simulation reflects the high impedance state while the logic analyzer depicts a partial return to the high impedance state from the address that last appeared on the bus (1004 - the address of the current instruction). The reasons for this difference have been explained and will not be further elaborated upon here.

MOVE.W D1,04(A1)

MOVE.W D1,04(A1) is a two-word instruction that uses the data register direct and register indirect with offset addressing modes to move the contents of data register D1 to the memory location pointed to by the sum of both address register A1 and the value of the instruction's extention word. The instruction consisted of two read cycles (displacement fetch and instruction prefetch) and one write cycle (move data). It required 15 clock cycles to execute. The four least significant address and data lines were monitored during the simulation. The results were:

```
                15-12 11-8 7--4 3--0 (columns)
                 AAAA DDDD AULR DFFF (signals)
                 4321 3210 SDD/ TCCC
                           SSW A210
                               C
                               K
                ---------------------
K(0) P(1)        1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)        0011 1111 1111 1010 $Address lines (12-15)
                                     $are 1006, location of
K(1) P(1)        0011 1111 0001 1010 $instruction being
K(1) P(2)        0011 1111 0001 1010 $prefetched

K(2) P(1)        0011 1111 0001 1010
K(2) P(2)        0011 1111 0001 1010
```

```
K(3) P(1)   0011 0100 0001 0010 $Data applied to data bus
K(3) P(2)   0011 0100 0001 0010 $(8-11). Data is 04,
                                $the displacement.
K(4) P(1)   0011 0100 0001 0010
K(4) P(2)   0011 0100 1111 1010 $End read cycle.

K(5) P(1)   1111 1111 1111 1010 $Begin read cycle.
K(5) P(2)   0100 1111 1111 1010 $Address lines (12-15)
                                $are 1008, location of
K(6) P(1)   0100 1111 0001 1010 $instruction being
K(6) P(2)   0100 1111 0001 1010 $prefetched.

K(7) P(1)   0100 1111 0001 1010
K(7) P(2)   0100 1111 0001 1010

K(8) P(1)   0100 0001 0001 0010 $Data applied to data bus
K(8) P(2)   0100 0001 0001 0010 $(8-11). Data is code
                                $for MOVE.W D1,08(A1).
K(9) P(1)   0100 0001 0001 0010
K(9) P(2)   0100 0001 1111 1010 $End read cycle.

K(10) P(1)  1111 1111 1111 1010 $Begin write cycle.
K(10) P(2)  0010 1111 1111 1001 $Address lines (12-15)
                                $are 2004.
K(11) P(1)  0010 1111 0110 1001
K(11) P(2)  0010 0101 0110 1001 $Data put on data bus
                                $(8-11). Data is 5 hex,
K(12) P(1)  0010 0101 0000 1001 $data being moved.
K(12) P(2)  0010 0101 0000 1001

K(13) P(1)  0010 0101 0000 1001
K(12) P(2)  0010 0101 0000 1001

K(13) P(1)  0010 0101 0000 0001
K(13) P(2)  0010 0101 0000 0001

K(14) P(1)  0010 0101 0000 0001
K(14) P(2)  0010 0101 1110 1001 $End write cycle.
```

The only difference between the simulation and the logic analyzer results occured on the address bus. At phase one of clock cycles zero and ten the logic analyzer catches the bus transitioning to the high impedance state whereas the simulation fully captures high impedance. This difference has been discussed earlier.

Also, for one and a half clock cycles beginning at

phase one of clock cycle 13, the logic analyzer data indicates that the address bus (A4-A1) is in state "0011" whereas on the previous cycle (phase two of clock cycle 12) it held the address of the memory location to which the data from D1 would be moved (2004). This address manifested itself as "0010" on A4-A1 (add a zero to catch PC<0>). A change to "0011" would indicate that the address was incremented to 2006 in the midst of a write cycle and then again returned to 2004 one and a half clock cycles later. Because this is abnormal behavior for the address bus during a write cycle, and it could not be duplicated, it appears to be a typing mistake.

MOVE.W D1,04(A1,D7)

MOVE.W D1,04(A1,D7) is also a two-word instruction that uses the data register direct and indexed register indirect with offset addressing modes to move the contents of data register D1 to the memory location determined by the sum of the index register (D7), the displacement (4), and the value of address register A1. It required 17 clock cycles to execute its two read and one write cycles. The same signals were monitored as with the previous instruction. The simulation results were:

```
15-12 11-8 7--4 3--0 (columns)
 AAAA DDDD AULR DFFF (signals)
 4321 3210 SDD/ TCCC
           SSW A210
               C
               K
--------------------
```

```
K(0) P(1)  1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)  0011 1111 1111 1010 $Address lines (12-15)
                               $are 1006, location of
K(1) P(1)  0011 1111 0001 1010 $instruction being
K(1) P(2)  0011 1111 0001 1010 $prefetched

K(2) P(1)  0011 1111 0001 1010
K(2) P(2)  0011 1111 0001 1010

K(3) P(1)  0011 0100 0001 0010 $Data applied to data bus
K(3) P(2)  0011 0100 0001 0010 $(8-11). Data is 04,
                               $the displacement.
K(4) P(1)  0011 0100 0001 0010
K(4) P(2)  0011 0100 1111 1010 $End read cycle.

K(5) P(1)  1111 1111 1111 1010 $Address and data
K(5) P(2)  1111 1111 1111 1010 $buses (8-15) go high.

K(6) P(1)  1111 1111 1111 1010
K(6) P(2)  1111 1111 1111 1010

K(7) P(1)  1111 1111 1111 1010 $Begin read cycle.
K(7) P(2)  0100 1111 1111 1010 $Address lines (12-15)
                               $are 1008, location of
K(8) P(1)  0100 1111 0001 1010 $instruction being
K(8) P(2)  0100 1111 0001 1010 $prefetched.

K(9) P(1)  0100 1111 0001 1010
K(9) P(2)  0100 1111 0001 1010

K(10) P(1) 0100 0001 0001 0010 $Data applied to data bus
K(10) P(2) 0100 0001 0001 0010 $(8-11). Data is code
                               $for MOVE.W D1,08(A1,D7).
K(11) P(1) 0100 0001 0001 0010
K(11) P(2) 0100 0001 1111 1010 $End read cycle.

K(12) P(1) 1111 1111 1111 1010 $Begin write cycle.
K(12) P(2) 0101 1111 1111 1001 $Address lines (12-15)
                               $are 200A.
K(13) P(1) 0101 1111 0110 1001
K(13) P(2) 0101 0101 0110 1001 $Data put on data bus
                               $(8-11). Data is 5 hex,
K(14) P(1) 0101 0101 0000 1001 $data being moved.
K(14) P(2) 0101 0101 0000 1001

K(15) P(1) 0101 0101 0000 1001
K(14) P(2) 0101 0101 0000 1001

K(15) P(1) 0101 0101 0000 0001
K(15) P(2) 0101 0101 0000 0001

K(16) P(1) 0101 0101 0000 0001
K(16) P(2) 0101 0101 1110 1001 $End write cycle.
```

The logic analyzer results indicate that the MC68000 returns both the address and data bus to the high impedance state at phase one of clock cycle five. This phase begins a two-cycle sequence of micro-operations that compute the destination address by adding both the displacement and the contents of the index register to the value contained in address register A1. The microstatements that return the bus to the high impedance state were not present in the CDL model. The assignments "ADENABLE = lo" and "DBENABLE = lo" were added to the ISP' model of this instruction to correct this omission.

There were also the usual deviations relative to the high impedance state on the address bus at the start of read and write cycles. These occurred at phase one of clock cycles zero and 12.

MOVE.W D1,$2004

MOVE.W D1,$2004 uses the data register direct and absolute short addressing modes to move the contents of data register D1 to the memory locations beginning at address 2004. This two-word instruction required 15 clock cycles to execute its two read and one write cycles. As with the previous instruction, the four least significant address and data lines were monitored. The simulation results were:

```
                  15-12 11-8 7--4 3--0 (columns)
                   AAAA DDDD AULR DFFF (signals)
                   4321 3210 SDD/ TCCC
                             SSW A210
                                 C
                                 K
                  --------------------
 K(0)  P(1)   1111 1111 1111 1010 $Begin read cycle.
 K(0)  P(2)   0011 1111 1111 1010 $Address lines (12-15)
                                  $are 1006, location of
 K(1)  P(1)   0011 1111 0001 1010 $instruction being
 K(1)  P(2)   0011 1111 0001 1010 $prefetched

 K(2)  P(1)   0011 1111 0001 1010
 K(2)  P(2)   0011 1111 0001 1010

 K(3)  P(1)   0011 0100 0001 0010 $Data applied to data bus
 K(3)  P(2)   0011 0100 0001 0010 $(8-11). Data is 04,
                                  $the displacement.
 K(4)  P(1)   0011 0100 0001 0010
 K(4)  P(2)   0011 0100 1111 1010 $End read cycle.

 K(5)  P(1)   1111 1111 1111 1010 $Begin read cycle.
 K(5)  P(2)   0100 1111 1111 1010 $Address lines (12-15)
                                  $are 1008, location of
 K(6)  P(1)   0100 1111 0001 1010 $instruction being
 K(6)  P(2)   0100 1111 0001 1010 $prefetched.

 K(7)  P(1)   0100 1111 0001 1010
 K(7)  P(2)   0100 1111 0001 1010

 K(8)  P(1)   0100 0001 0001 0010 $Data applied to data bus
 K(8)  P(2)   0100 0001 0001 0010 $(8-11). Data is code
                                  $for MOVE.W D1,$2008.
 K(9)  P(1)   0100 0001 0001 0010
 K(9)  P(2)   0100 0001 1111 1010 $End read cycle.

 K(10) P(1)   1111 1111 1111 1010 $Begin write cycle.
 K(10) P(2)   0010 1111 1111 1001 $Address lines (12-15)
                                  $are 2004.
 K(11) P(1)   0010 1111 0110 1001
 K(11) P(2)   0010 0101 0110 1001 $Data put on data bus
                                  $(8-11). Data is 5 hex,
 K(12) P(1)   0010 0101 0000 1001 $data being moved.
 K(12) P(2)   0010 0101 0000 1001

 K(13) P(1)   0010 0101 0000 1001
 K(12) P(2)   0010 0101 0000 1001

 K(13) P(1)   0010 0101 0000 0001
 K(13) P(2)   0010 0101 0000 0001

 K(14) P(1)   0010 0101 0000 0001
```

```
K(14) P(2)  0010 0101 1110 1001 $End write cycle.
```

The only differences occurred at phase one of clock cycles zero and ten where the logic analyzer again did not let the address bus complete its return to the high impedance state.

MOVE.W A1,D3

MOVE.W A1,D3 is a single-word instruction that uses the address register direct and data register direct addressing modes to move the contents of address register A1 to data register D3. It consists of a single read cycle (instruction prefetch) that required five clock cycles to execute. No address lines were monitored during this simulation. The results were:

```
                15-12 11-8 7--4 3--0 (columns)
                 DDDD DDDD AULR DFFF (signals)
                 7654 3210 SDD/ TCCC
                           SSW A210
                               C
                               K
                ---------------------
K(0) P(1)  1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)  1111 1111 1111 1010

K(1) P(1)  1111 1111 0001 1010
K(1) P(2)  1111 1111 0001 1010

K(2) P(1)  1111 1111 0001 1010
K(2) P(2)  1111 1111 0001 1010

K(3) P(1)  0000 1001 0001 0010 $Data applied to data bus
K(3) P(2)  0000 1001 0001 0010 $(8-15). Data is code
                               $for MOVE.W A1,D3.
K(4) P(1)  0000 1001 0001 0010
K(4) P(2)  0000 1001 1111 1010 $End read cycle.
```

At phase one of clock cycle zero of their CDL model,

Hamby and Guillory included the microstatement "IDBUS <- A1(LWORD)" to place the low word of address register A1 onto the internal data bus. Subregister A1(LWORD) did not appear in the declaration section of their CDL model but was added to the ISP' version of this instruction.

At phase two of clock cycle two the logic analyzer shows the data bus changed from the high impedance state of the previous cycle to "1011 1111" on its way to the valid data state "0000 1001" at phase one of clock cycle three. As occurred during the MOVE.L D1,A1 simulation, the logic analyzer caught the bus transitioning to high impedance whereas the simulation shows the process completed.

MOVE.W (A1),D2

MOVE.W (A1),D2 uses the address register indirect and data register direct addressing modes to move the contents of the memory locations pointed to by address register A1 to data register D2. This single-word instruction required 10 clock cycles to execute its two read cycles. Both the simulation and logic analyzer results were in agreement. No address lines were monitored during the simulation. The results were:

```
                15-12 11-8 7--4 3--0 (columns)
                 DDDD DDDD AULR DFFF (signals)
                 7654 3210 SDD/ TCCC
                           SSW A210
                               C
                               K
                ---------------------
K(0) P(1)        1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)        1111 1111 1111 1010
```

```
K(1) P(1)   1111 1111 0001 1010
K(1) P(2)   1111 1111 0001 1010

K(2) P(1)   1111 1111 0001 1010
K(2) P(2)   1111 1111 0001 1010

K(3) P(1)   0001 0001 0001 0010 $Data applied to data bus
K(3) P(2)   0001 0001 0001 0010 $(8-15). Data is code
                                $for MOVE.W (A1),D2.
K(4) P(1)   0001 0001 0001 0010
K(4) P(2)   0001 0001 1111 1010 $End read cycle.

K(5) P(1)   1111 1111 1111 1010 $Begin read cycle.
K(5) P(2)   1111 1111 1111 1001 $Function codes (0-2)
                                $change to user data
K(6) P(1)   1111 1111 0001 1001 $mode.
K(6) P(2)   1111 1111 0001 1001

K(7) P(1)   1111 1111 0001 1001
K(7) P(2)   1111 1111 0001 1001

K(8) P(1)   0101 0101 0001 0001 $Data applied to data bus
K(8) P(2)   0101 0101 0001 0001 $(8-15). Data is 55 hex,
                                $data being moved.
K(9) P(1)   0101 0101 0001 0001
K(9) P(2)   0101 0101 1111 1001 $End read cycle.
```

MOVE.W (A1)+,D6

MOVE.W (A1)+,D6 uses the postincrement register indirect and data register direct addressing modes to move the contents of the memory locations pointed to by address register A1 to data register D6. Address register A1 is incremented by two after being used as a pointer. This single-word instruction required 10 clock cycles to execute its two read cycles. The four least significant address and data lines were monitored during the simulation. The results were:

```
                15-12 11-8 7--4 3--0 (columns)
                 AAAA DDDD AULR DFFF (signals)
                 4321 3210 SDD/ TCCC
                           SSW A210
                               C
                               K
                ---------------------
K(0) P(1)  1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)  0011 1111 1111 1010 $Address lines (12-15)
                               $are 1006, location of
K(1) P(1)  0011 1111 0001 1010 $instruction being
K(1) P(2)  0011 1111 0001 1010 $prefetched

K(2) P(1)  0011 1111 0001 1010
K(2) P(2)  0011 1111 0001 1010

K(3) P(1)  0011 1001 0001 0010 $Data applied to data bus
K(3) P(2)  0011 1001 0001 0010 $(8-11). Data is code
                               $for MOVE.W (A1)+,D7.
K(4) P(1)  0011 1001 0001 0010
K(4) P(2)  0011 1001 1111 1010 $End read cycle.

K(5) P(1)  1111 1111 1111 1010 $Begin read cycle.
K(5) P(2)  0000 1111 1111 1001 $Address lines (12-15)
                               $are 2000 hex, location of
K(6) P(1)  0000 1111 0001 1001 $of data. Function codes
K(6) P(2)  0000 1111 0001 1001 $(0-2) change to user
                               $data mode.
K(7) P(1)  0000 1111 0001 1001
K(7) P(2)  0000 1111 0001 1001

K(8) P(1)  0000 0101 0001 0001 $Data applied to data bus
K(8) P(2)  0000 0101 0001 0001 $(8-11). Data is 5 hex,
                               $data being moved.
K(9) P(1)  0000 0101 0001 0001
K(9) P(2)  0000 0101 1111 1001 $End read cycle.
```

The simulation results again differed from the logic analyzer data only on the address bus where the logic analyzer failed to capture its high impedance state at phase one of clock cycle zero and five.

MOVE.W -(A1),D4

MOVE.W -(A1),D4 uses the predecrement register indirect and data register direct addressing modes to move the

contents of the memory locations pointed to by address register A1 to data register D4. Address register A1 is decremented by two before being used as a pointer. This single-word instruction required 12 clock cycles to execute its two read cycles. The four least significant address and data lines were monitored during the simulation. The results were:

```
                  15-12 11-8 7--4 3--0 (columns)
                   AAAA DDDD AULR DFFF (signals)
                   4321 3210 SDD/ TCCC
                             SSW A210
                                 C
                                 K
                  --------------------
 K(0) P(1)   1111 1111 1111 1010 $Begin read cycle.
 K(0) P(2)   0011 1111 1111 1010 $Address lines (12-15)
                                 $are 1006, location of
 K(1) P(1)   0011 1111 0001 1010 $instruction being
 K(1) P(2)   0011 1111 0001 1010 $prefetched

 K(2) P(1)   0011 1111 0001 1010
 K(2) P(2)   0011 1111 0001 1010

 K(3) P(1)   0011 0001 0001 0010 $Data applied to data bus
 K(3) P(2)   0011 0001 0001 0010 $(8-11). Data is code
                                 $for MOVE.W -(A1),D3.
 K(4) P(1)   0011 0001 0001 0010
 K(4) P(2)   0011 0001 1111 1010 $End read cycle.

 K(5) P(1)   1111 1111 1111 1010 $Address and data
 K(5) P(2)   1111 1111 1111 1010 $buses (8-15) go high.

 K(6) P(1)   1111 1111 1111 1010
 K(6) P(2)   1111 1111 1111 1010

 K(7) P(1)   1111 1111 1111 1010 $Begin read cycle.
 K(7) P(2)   0011 1111 1111 1001 $Address lines (12-15)
                                 $are 2006, location of
 K(8) P(1)   0011 1111 0001 1001 $data. Function codes
 K(8) P(2)   0011 1111 0001 1001 $(0-2) change to user
                                 $data mode.
 K(9) P(1)   0011 1111 0001 1001
 K(9) P(2)   0011 1111 0001 1001

K(10) P(1)   0011 0101 0001 0001 $Data applied to data bus
```

```
K(10) P(2)   0011 0101 0001 0001 $(8-11). Data is 5 hex,
                                  $data being moved.
K(11) P(1)   0011 0101 0001 0001
K(11) P(2)   0011 0101 1111 1001 $End read cycle.
```

The logic analyzer results indicate that the MC68000 returns both the address and data bus to the high impedance state at phase one of clock cycle five. This phase begins a two-cycle sequence in which address register A1 is decremented. The microstatements that return the bus to the high impedance state were not present in the CDL model. The assignments "ADENABLE = lo" and "DBENABLE = lo" were added to the ISP' model of this instruction to correct this omission.

There were also the usual deviations relative to the high impedance state on the address bus at the start of read and write cycles. These occurred at phase one of clock cycles zero and 5.

MOVE.W 04(A1),D1

The MOVE.W 04(A1),D1 instruction uses the data register direct and register indirect with offset addressing modes to move the contents of the memory locations determined by the sum of the displacement (4 - located in the instruction's extention word) and the contents of address register A1. This two-word instruction required 15 clock cycles to execute its three read cycles. No address lines were monitored during the simulation. The results were:

```
                 15-12 11-8 7--4 3--0 (columns)
                  DDDD DDDD AULR DFFF (signals)
                  7654 3210 SDD/ TCCC
                            SSW A210
                               C
                               K
                 ------------------
 K(0)  P(1)       1111 1111 1111 1010 $Begin read cycle.
 K(0)  P(2)       1111 1111 1111 1010

 K(1)  P(1)       1111 1111 0001 1010
 K(1)  P(2)       1111 1111 0001 1010

 K(2)  P(1)       1111 1111 0001 1010
 K(2)  P(2)       1111 1111 0001 1010

 K(3)  P(1)       0000 0100 0001 0010 $Data applied to data bus
 K(3)  P(2)       0000 0100 0001 0010 $(8-15). Data is 4 hex,
                                      $the displacement.
 K(4)  P(1)       0000 0100 0001 0010
 K(4)  P(2)       0000 0100 1111 1010 $End read cycle.

 K(5)  P(1)       1111 1111 1111 1010 $Begin read cycle.
 K(5)  P(2)       1111 1111 1111 1010

 K(6)  P(1)       1111 1111 0001 1010
 K(6)  P(2)       1111 1111 0001 1010

 K(7)  P(1)       1111 1111 0001 1010
 K(7)  P(2)       1111 1111 0001 1010

 K(8)  P(1)       0010 1001 0001 0010 $Data applied to data bus
 K(8)  P(2)       0010 1001 0001 0010 $(8-15). Data is code
                                      $for MOVE.W 08(A1),D2.
 K(9)  P(1)       0010 1001 0001 0010
 K(9)  P(2)       0010 1001 1111 1010 $End read cycle.

 K(10) P(1)       1111 1111 1111 1010 $Begin read cycle.
 K(10) P(2)       1111 1111 1111 1001 $Function codes (0-2)
                                      $change to user data
 K(11) P(1)       1111 1111 0001 1001 $mode.
 K(11) P(2)       1111 1111 0001 1001

 K(12) P(1)       1111 1111 0001 1001
 K(12) P(2)       1111 1111 0001 1001

 K(13) P(1)       0101 0101 0001 0001 $Data applied to data bus
 K(13) P(2)       0101 0101 0001 0001 $(8-15). Data is 55 hex,
                                      $data being moved.
 K(14) P(1)       0101 0101 0001 0001
 K(14) P(2)       0101 0101 1111 1001 $End read cycle.
```

The only difference between the simulation and logic analyzer data appeared on the data bus at phase two of clock cycle two. Here the logic analyzer caught the bus transitioning to a valid data state from high impedance whereas the simulation maintained high impedance.

MOVE.W 04(A1,D7),D2

MOVE.W 04(A1,D7),D2 is a two-word instruction that uses the indexed register indirect with offset and data register direct addressing modes to move the contents of the memory locations determined by the sum of the index register D7, the displacement (4), and the contents of address register A1, to data register D2. This instruction required 17 clock cycles to complete its three read cycles. The four least significant address and data lines were monitored during the simulation. The results were:

```
                 15-12 11-8 7--4 3--0 (columns)
                  AAAA DDDD AULR DFFF (signals)
                  4321 3210 SDD/ TCCC
                            SSW A210
                                C
                                K
                 --------------------
K(0) P(1)          1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)          0011 1111 1111 1010 $Address lines (12-15)
                                       $are 1006, location of
K(1) P(1)          0011 1111 0001 1010 $instruction being
K(1) P(2)          0011 1111 0001 1010 $prefetched

K(2) P(1)          0011 1111 0001 1010
K(2) P(2)          0011 1111 0001 1010

K(3) P(1)          0011 0100 0001 0010 $Data applied to data bus
K(3) P(2)          0011 0100 0001 0010 $(8-11). Data is 4 hex,
                                       $the displacement.
K(4) P(1)          0011 0100 0001 0010
K(4) P(2)          0011 0100 1111 1010 $End read cycle.
```

```
 K(5) P(1)  1111 1111 1111 1010 $Address and data
 K(5) P(2)  1111 1111 1111 1010 $buses (8-15) go high.

 K(6) P(1)  1111 1111 1111 1010
 K(6) P(2)  1111 1111 1111 1010

 K(7) P(1)  1111 1111 1111 1010 $Begin read cycle.
 K(7) P(2)  0100 1111 1111 1010 $Address lines (12-15)
                                $are 1008, location of
 K(8) P(1)  0100 1111 0001 1010 $instruction being
 K(8) P(2)  0100 1111 0001 1010 $prefetched.

 K(9) P(1)  0100 1111 0001 1010
 K(9) P(2)  0100 1111 0001 1010

K(10) P(1)  0100 0001 0001 0010 $Data applied to data bus
K(10) P(2)  0100 0001 0001 0010 $(8-11). Data is code
                                $for MOVE.W 04(A1,D7),D3.
K(11) P(1)  0100 0001 0001 0010
K(11) P(2)  0100 0001 1111 1010 $End read cycle.

K(12) P(1)  1111 1111 1111 1010 $Begin read cycle.
K(12) P(2)  0101 1111 1111 1001 $Address lines (12-15)
                                $are 200A, location of
K(13) P(1)  0101 1111 0001 1001 $of data. Function codes
K(13) P(2)  0101 1111 0001 1001 $(0-2) change to user
                                $data mode.
K(14) P(1)  0101 1111 0001 1001
K(14) P(2)  0101 1111 0001 1001

K(15) P(1)  0101 0101 0001 0001 $Data applied to data bus
K(15) P(2)  0101 0101 0001 0001 $(8-11). Data is 5 hex,
                                $data being moved.
K(16) P(1)  0101 0101 0001 0001
K(16) P(2)  0101 0101 1111 1001 $End read cycle.
```

The logic analyzer results indicate that the MC68000 returns both the address and data bus to the high impedance state at phase one of clock cycle five. This phase begins a two-cycle sequence in which the index register D7 is added to the displacement register (DISREG). The microstatements that return the bus to the high impedance state were not present in the CDL model. The assignments "ADENABLE = lo" and "DBENABLE = lo" were added to the ISP' model of this instruction to correct this omission.

There were also the usual deviations relative to the high impedance state on the address bus at the start of read and write cycles. These occurred at phase one of clock cycles zero and 12.

MOVE.W $2004,D5

MOVE.W $2004,D5 uses the absolute short and data register direct addressing modes to move the data word at memory location 2004 into data register D5. This two-word instruction required 15 clock cycles to complete its three read cycles. The four least significant address and data signals were monitored during the simulation. The results were:

```
                15-12 11-8 7--4 3--0 (columns)
                 AAAA DDDD AULR DFFF (signals)
                 4321 3210 SDD/ TCCC
                           SSW A210
                               C
                               K
                ---------------------
K(0) P(1)        1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)        0011 1111 1111 1010 $Address lines (12-15)
                                     $are 1006, location of
K(1) P(1)        0011 1111 0001 1010 $instruction being
K(1) P(2)        0011 1111 0001 1010 $prefetched

K(2) P(1)        0011 1111 0001 1010
K(2) P(2)        0011 1111 0001 1010

K(3) P(1)        0011 0100 0001 0010 $Data applied to data bus
K(3) P(2)        0011 0100 0001 0010 $(8-11). Data is 4 hex,
                                     $low byte of operand
K(4) P(1)        0011 0100 0001 0010 $address.
K(4) P(2)        0011 0100 1111 1010 $End read cycle.

K(5) P(1)        1111 1111 1111 1010 $Begin read cycle.
K(5) P(2)        0100 1111 1111 1010 $Address lines (12-15)
                                     $are 1008, location of
K(6) P(1)        0100 1111 0001 1010 $instruction being
K(6) P(2)        0100 1111 0001 1010 $prefetched.
```

```
K(7) P(1)   0100 1111 0001 1010
K(7) P(2)   0100 1111 0001 1010

K(8) P(1)   0100 1000 0001 0010 $Data applied to data bus
K(8) P(2)   0100 1000 0001 0010 $(8-11). Data is code
                                $for MOVE.W $2004,D6.
K(9) P(1)   0100 1000 0001 0010
K(9) P(2)   0100 1000 1111 1010 $End read cycle.

K(10) P(1)  1111 1111 1111 1010 $Begin read cycle.
K(10) P(2)  0010 1111 1111 1001 $Address lines (12-15)
                                $are 2004, location of
K(11) P(1)  0010 1111 0001 1001 $of data. Function codes
K(11) P(2)  0010 1111 0001 1001 $(0-2) change to user
                                $data mode.
K(12) P(1)  0010 1111 0001 1001
K(12) P(2)  0010 1111 0001 1001

K(13) P(1)  0010 0101 0001 0001 $Data applied to data bus
K(13) P(2)  0010 0101 0001 0001 $(8-11). Data is 5 hex,
                                $data being moved.
K(14) P(1)  0010 0101 0001 0001
K(14) P(2)  0010 0101 1111 1001 $End read cycle.
```

The usual deviations relative to the high impedance state on the address bus at the start of a read cycle occurred at phase one of clock cycles zero and 10.

MOVE.W $2004,$2008

MOVE.W $2008,$2004 uses the absolute long addressing mode to move the data word beginning at memory location 2004 to 2008. This instruction is five words long and requires 35 clock cycles to complete its six read and one write cycles. No address lines were monitored. The simulation results were:

```
15-12 11-8 7--4 3--0 (columns)
 DDDD DDDD AULR DFFF (signals)
 7654 3210 SDD/ TCCC
           SSW A210
               C
               K
```

```
                 ------------------
K(0)  P(1)   1111 1111 1111 1010 $Begin read cycle.
K(0)  P(2)   1111 1111 1111 1010

K(1)  P(1)   1111 1111 0001 1010
K(1)  P(2)   1111 1111 0001 1010

K(2)  P(1)   1111 1111 0001 1010
K(2)  P(2)   1111 1111 0001 1010

K(3)  P(1)   0000 0000 0001 0010 $Data applied to data bus
K(3)  P(2)   0000 0000 0001 0010 $(8-15). Data is high
                                 $word of source address.
K(4)  P(1)   0000 0000 0001 0010
K(4)  P(2)   0000 0000 1111 1010 $End read cycle.

K(5)  P(1)   1111 1111 1111 1010 $Begin read cycle.
K(5)  P(2)   1111 1111 1111 1010

K(6)  P(1)   1111 1111 0001 1010
K(6)  P(2)   1111 1111 0001 1010

K(7)  P(1)   1111 1111 0001 1010
K(7)  P(2)   1111 1111 0001 1010

K(8)  P(1)   0000 0100 0001 0010 $Data applied to data bus
K(8)  P(2)   0000 0100 0001 0010 $(8-15). Data is low
                                 $word of source address.
K(9)  P(1)   0000 0100 0001 0010
K(9)  P(2)   0000 0100 1111 1010 $End read cycle.

K(10) P(1)   1111 1111 1111 1010 $Begin read cycle.
K(10) P(2)   1111 1111 1111 1010

K(11) P(1)   1111 1111 0001 1010
K(11) P(2)   1111 1111 0001 1010

K(12) P(1)   1111 1111 0001 1010
K(12) P(2)   1111 1111 0001 1010

K(13) P(1)   0000 0000 0001 0010 $Data applied to data bus
K(13) P(2)   0000 0000 0001 0010 $(8-15). Data is high
                                 $word of destination.
K(14) P(1)   0000 0000 0001 0010
K(14) P(2)   0000 0000 1111 1010 $End read cycle.

K(15) P(1)   1111 1111 1111 1010 $Begin read cycle.
K(15) P(2)   1111 1111 1111 1001 $Function codes (0-2)
                                 $change to user data
K(16) P(1)   1111 1111 0001 1001 $mode.
K(16) P(2)   1111 1111 0001 1001

K(17) P(1)   1111 1111 0001 1001
K(17) P(2)   1111 1111 0001 1001
```

```
K(18) P(1)  0101 0101 0001 0001 $Data applied to data bus
K(18) P(2)  0101 0101 0001 0001 $(8-15). Data is 55 hex,
                                $data being moved.
K(19) P(1)  0101 0101 0001 0001
K(19) P(2)  0101 0101 1111 1001 $End read cycle.

K(20) P(1)  1111 1111 1111 1001 $Begin read cycle.
K(20) P(2)  1111 1111 1111 1010 $Function codes (0-2)
                                $change to user program
K(21) P(1)  1111 1111 0001 1010 $mode.
K(21) P(2)  1111 1111 0001 1010

K(22) P(1)  1111 1111 0001 1010
K(22) P(2)  1111 1111 0001 1010

K(23) P(1)  0000 1000 0001 0010 $Data applied to data bus
K(23) P(2)  0000 1000 0001 0010 $(8-15). Data is low word
                                $of destination.
K(24) P(1)  0000 1000 0001 0010
K(24) P(2)  0000 1000 1111 1010 $End read cycle.

K(25) P(1)  1111 1111 1111 1010 $Begin write cycle.
K(25) P(2)  1111 1111 1111 1001 $Function codes (0-2)
                                $change to user data mode.
K(26) P(1)  1111 1111 0110 1001
K(26) P(2)  0101 0101 0110 1001 $Data put on data bus
                                $(8-15). Data is 55 hex,
K(27) P(1)  0101 0101 0000 1001 $data being moved.
K(27) P(2)  0101 0101 0000 1001

K(28) P(1)  0101 0101 0000 1001
K(27) P(2)  0101 0101 0000 1001

K(28) P(1)  0101 0101 0000 0001
K(28) P(2)  0101 0101 0000 0001

K(29) P(1)  0101 0101 0000 0001
K(29) P(2)  0101 0101 1110 1001 $End write cycle.

K(30) P(1)  1111 1111 1111 1001 $Begin read cycle.
K(30) P(2)  1111 1111 1111 1010 $Function codes (0-2)
                                $change to user program
K(31) P(1)  1111 1111 0001 1010 $mode.
K(31) P(2)  1111 1111 0001 1010

K(32) P(1)  1111 1111 0001 1010
K(32) P(2)  1111 1111 0001 1010

K(33) P(1)  1101 0000 0001 0010 $Data applied to data bus
K(33) P(2)  1101 0000 0001 0010 $(8-15). Data is code
                                $for JMP, so this is
K(34) P(1)  1101 0000 0001 0010 $a prefetch.
K(34) P(2)  1101 0000 1111 1010 $End read cycle.
```

All differences between the simulation and logic analyzer data occur on the data bus. At phase two of clock cycles 2, 7, 12, 22, and 26 the logic analyzer catches the data bus transitioning to a valid data state. There is also a difference at phase one of clock cycle 30. Because this cycle follows a write cycle, the data buses'return to the high impedance state is not captured until phase two.

MOVE.W #$5555,D1

MOVE.W #$5555,D1 uses the immediate and data register direct addressing modes to move the hex value 5555 into data register D1. This is a two-word instruction that requires 10 clock cycles to complete its two read cycles. The four least significant signals on the address and data bus were monitored during the simulation. The results were:

```
                 15-12 11-8 7--4 3--0 (columns)
                  AAAA DDDD AULR DFFF (signals)
                  4321 3210 SDD/ TCCC
                            SSW A210
                                C
                                K
                 ---------------------
K(0) P(1)   1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)   0011 1111 1111 1010 $Address lines (12-15)
                                $are 1006, location of
K(1) P(1)   0011 1111 0001 1010 $immediate data.
K(1) P(2)   0011 1111 0001 1010

K(2) P(1)   0011 1111 0001 1010
K(2) P(2)   0011 1111 0001 1010

K(3) P(1)   0011 0101 0001 0010 $Data applied to data bus
K(3) P(2)   0011 0101 0001 0010 $(8-11). Data is 5 hex,
                                $immediate data.
K(4) P(1)   0011 0101 0001 0010
K(4) P(2)   0011 0101 1111 1010 $End read cycle.

K(5) P(1)   1111 1111 1111 1010 $Begin read cycle.
```

```
K(5) P(2)   0100 1111 1111 1010 $Address lines (12-15)
                                 $are 1008, location of
K(6) P(1)   0100 1111 0001 1010 $instruction being
K(6) P(2)   0100 1111 0001 1010 $prefetched.

K(7) P(1)   0100 1111 0001 1010
K(7) P(2)   0100 1111 0001 1010

K(8) P(1)   0100 1100 0001 0010 $Data applied to data bus
K(8) P(2)   0100 1100 0001 0010 $(8-11). Data is code
                                 $for MOVE.W #$5555,D1.
K(9) P(1)   0100 1100 0001 0010
K(9) P(2)   0100 1100 1111 1010 $End read cycle.
```

A difference occurs at phase one of clock cycle zero where the logic analyzer captured the address bus transitioning to the high impedance state whereas the simulation fully reflects high impedance. Also, at phase one of clock cycle nine, the logic analyzer data shows a data bus state change where one should clearly not occur. This is certainly a typing error.

JMP (A0)

JMP (A0) uses the register indirect addressing mode to direct the MC68000 to next execute the instruction located at the address pointed to by address register A0. This instruction ended all instruction test routines creating a loop in which the instruction of interest was continually executed. JMP is a single-word instruction that requires 10 clock cycles to execute its two read cycles (instruction prefetches). The simulation and logic analyzer results were equivalent. No address lines were monitored during the simulation. The results were:

```
                15-12 11-8 7--4 3--0 (columns)
                 DDDD DDDD AULR DFFF (signals)
                 7654 3210 SDD/ TCCC
                           SSW A210
                              C
                              K
               ---------------------
K(0) P(1)       1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)       1111 1111 1111 1010

K(1) P(1)       1111 1111 0001 1010
K(1) P(2)       1111 1111 0001 1010

K(2) P(1)       1111 1111 0001 1010
K(2) P(2)       1111 1111 0001 1010

K(3) P(1)       1111 1111 0001 0010 $Data applied to data bus
K(3) P(2)       1111 1111 0001 0010 $(8-15). Data is all ones
                                    $because this a prefetch
K(4) P(1)       1111 1111 0001 0010 $and there are no instruct-
                                    $ions following the JMP
K(4) P(2)       1111 1111 1111 1010 $(unused memory is all 1's).

K(5) P(1)       1111 1111 1111 1010 $Begin read cycle.
K(5) P(2)       1111 1111 1111 1010

K(6) P(1)       1111 1111 0001 1010
K(6) P(2)       1111 1111 0001 1010

K(7) P(1)       1111 1111 0001 1010
K(7) P(2)       1111 1111 0001 1010

K(8) P(1)       0000 0001 0001 0010 $Data applied to data bus
K(8) P(2)       0000 0001 0001 0010 $(8-15). Data is code for
                                    $MOVE.W D1,D2, so this is a
K(9) P(1)       0000 0001 0001 0010 $fetch.
K(9) P(2)       0000 0001 1111 1010 $End read cycle.
```

ADD.W D3,D5

ADD.W D3,D5 uses the data register direct addressing mode to sum the contents of data registers D3 and D5 and then store the result in D5. It is a single-word instruction requiring five clock cycles to execute its read cycle (prefetch). No address lines were monitored during its simulation. The results were:

```
                  15-12 11-8 7--4 3--0 (columns)
                   DDDD DDDD AULR DFFF (signals)
                   7654 3210 SDD/ TCCC
                             SSW A210
                                 C
                                 K
                  --------------------
K(0) P(1)   1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)   1111 1111 1111 1010

K(1) P(1)   1111 1111 0001 1010
K(1) P(2)   1111 1111 0001 1010

K(2) P(1)   1111 1111 0001 1010
K(2) P(2)   1111 1111 0001 1010

K(3) P(1)   1000 0101 0001 0010 $Data applied to data bus
K(3) P(2)   1000 0101 0001 0010 $(8-15). Data is code for
                                $MOVE.W D5,(A2).
K(4) P(1)   1000 0101 0001 0010
K(4) P(2)   1000 0101 1111 1010 $End read cycle.
```

The simulation and data logic analyzer results differed only on the data bus at phase two of clock cycle two. Here the logic analyzer caught the data bus transitioning to a valid data state while the simulation maintained high impedance.

BEQ START

BEQ (Branch If Equal) is one of 14 variations of the MC68000's unconditional branch instructions. If the status register's zero condition code bit is set, then the program's execution will branch and continue at the location identified by the label "START"; otherwise, program execution continues with the instruction immediately following the BEQ instruction.

Hamby and Guillory tested the instruction "BEQ $1004" where 1004 was the address of the first substantive

instruction of the test routine (MOVE.W D1,D3). However, the simulation uses the label "START" to identify the location of the instruction. The location of this instruction differed in the simulation (1000) because its test routine did not include as its first instruction "AND.W #$DFFF,SR" to initialize the status register. This was accomplished in the ISP' initialization routine for this instruction model. This difference did not appear in the data being compared because the address lines were not monitored.

The simulation test routine examined processing in the case where the branch was taken as well as when it was not. The single-word instruction's execution time is dependent upon whether the branch is taken. If the branch condition is true, then the instruction required 12 clock cycles to complete its two read cycles; otherwise, it is completed in nine (single read cycle).

The CDL model contained two discrepancies. At phase two of clock cycle four, the program counter is incremented by two with the microstatement "PC <- PCadd2". This should not occur at this point because if the branch is taken the program counter will again be adjusted by the instruction's displacement at phase one of clock cycle nine (PC <- PCaddIR(0-7)). This could be properly handled if the compiler is aware that the program counter will be two words beyond the BEQ instruction at the time of the branch and computes the displacement accordingly. But to remain consistent with previous micro-operations, the program

counter can best be incremented at phase two of clock cycle eight where it will be incremented only when the branch is not taken.

The microstatement "IR <- PFR" was added to those occurring at phase two of clock cycle 15. At this point, the branch has been taken and the instruction branched to has been fetched into the prefetch register. The contents of the prefetch register in turn must be placed in the instruction register so that the instruction can be decoded for execution. Alternately, since this is actually a fetch rather than a prefetch, instead of placing the contents of the external data buffer into the prefetch register with the statement "PFR <- EXDBUF" at phase one of clock cycle 15, its contents could be placed directly into the instruction register with the statement "IR <- EXDBUF" with equivalent results.

The only difference between the simulation and logic analyzer results again occurred when the logic analyzer caught the data bus transitioning to a valid data state at phase two of clock cycle two and 13. The complete results were:

Analyzer output (condition true, branch taken):

```
                15-12 11-8 7--4 3--0 (columns)
                 DDDD DDDD AULR DFFF (signals)
                 7654 3210 SDD/ TCCC
                           SSW A210
                               C
                               K
                ---------------------
  K(0) P(1)      1111 1111 1111 1010 $Begin read cycle.
```

```
K(0) P(2)  1111 1111 1111 1010

K(1) P(1)  1111 1111 0001 1010
K(1) P(2)  1111 1111 0001 1010

K(2) P(1)  1111 1111 0001 1010
K(2) P(2)  1111 1111 0001 1010

K(3) P(1)  1101 0000 0001 0010 $Data applied to data bus
K(3) P(2)  1101 0000 0001 0010 $(8-15). Data is code
                               $for JMP (A0).
K(4) P(1)  1101 0000 0001 0010
K(4) P(2)  1101 0000 1111 1010 $End read cycle.

K(9) P(1)  1111 1111 1111 1010 $Data bus (8-15)
K(9) P(2)  1111 1111 1111 1010 $goes high.

K(10) P(1) 1111 1111 1111 1010
K(10) P(2) 1111 1111 1111 1010

K(11) P(1) 1111 1111 0001 1010 $Begin read cycle.
K(11) P(2) 1111 1111 0001 1010

K(12) P(1) 1111 1111 0001 1010
K(12) P(2) 1111 1111 0001 1010

K(13) P(1) 1111 1111 0001 1010
K(13) P(2) 1111 1111 0001 1010

K(14) P(1) 0000 0001 0001 0010 $Data applied to data bus
K(14) P(2) 0000 0001 0001 0010 $(8-15). Data is code for
                               $MOVE.W D1,D3, so this is a
K(15) P(1) 0000 0001 0001 0010 $fetch.
K(15) P(2) 0000 0001 1111 1010 $End read cycle.
```

Analyzer output (condition false, branch not taken):

```
          15-12 11-8 7--4 3--0 (columns)
           DDDD DDDD AULR DFFF (signals)
           7654 3210 SDD/ TCCC
                      SSW A210
                          C
                          K
          --------------------
K(0) P(1)  1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)  1111 1111 1111 1010

K(1) P(1)  1111 1111 0001 1010
K(1) P(2)  1111 1111 0001 1010

K(2) P(1)  1111 1111 0001 1010
K(2) P(2)  1111 1111 0001 1010

K(3) P(1)  0000 0010 0001 0010 $Data applied to data bus
K(3) P(2)  0000 0010 0001 0010 $(8-15). Data is code
```

```
                                                $for MOVE.W D2,D3.
K(4) P(1)   0000 0010 0001 0010
K(4) P(2)   0000 0010 1111 1010 $End read cycle.

K(5) P(1)   1111 1111 1111 1010 $Data bus (8-15)
K(5) P(2)   1111 1111 1111 1010 $goes high.

K(6) P(1)   1111 1111 1111 1010
K(6) P(2)   1111 1111 1111 1010

K(7) P(1)   1111 1111 1111 1010
K(7) P(2)   1111 1111 1111 1010

K(8) P(1)   1111 1111 1111 1010
K(8) P(2)   1111 1111 1111 1010 $End BEQ.
```

BTST D1,(A1)

The BTST D1,(A1) instruction will direct the MC68000 to retrieve a byte from the memory location specified by address register A1 and then test the bit identified by the contents of data register D1. If the selected bit is zero, then the status register's zero condition code bit (Z) is set; it is otherwise cleared. This single-word instruction required 10 clock cycles to execute its two read cycles. No address signals were monitored during the simulation. The results were:

```
                15-12 11-8 7--4 3--0 (columns)
                 DDDD DDDD AULR DFFF (signals)
                 7654 3210 SDD/ TCCC
                           SSW A210
                               C
                               K
               ---------------------
K(0) P(1)   1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)   1111 1111 1111 1010

K(1) P(1)   1111 1111 0001 1010
K(1) P(2)   1111 1111 0001 1010

K(2) P(1)   1111 1111 0001 1010
K(2) P(2)   1111 1111 0001 1010
```

```
K(3) P(1)  0000 0010 0001 0010 $Data applied to data bus
K(3) P(2)  0000 0010 0001 0010 $(8-15). Data is code
                               $for MOVE.W D2,D3 so this
K(4) P(1)  0000 0010 0001 0010 $is a prefetch.
K(4) P(2)  0000 0010 1111 1010 $End read cycle.

K(5) P(1)  1111 1111 1111 1010 $Begin read cycle.
K(5) P(2)  1111 1111 1111 1001

K(6) P(1)  1111 1111 0101 1001 $UDS' (6) not asserted
K(6) P(2)  1111 1111 0101 1001 $so only the low byte
                               $is read from memory.
K(7) P(1)  1111 1111 0101 1001
K(7) P(2)  1111 1111 0101 1001

K(8) P(1)  0101 0101 0101 0001 $Data applied to data bus
K(8) P(2)  0101 0101 0101 0001 $(8-15). Data is 55 hex.

K(9) P(1)  0101 0101 0101 0001
K(9) P(2)  0101 0101 1111 1001 $End read cycle.
```

The simulation and logic analyzer results differed on the data bus at phase two of clock cycle two. Again, the logic analyzer caught the bus transitioning to a valid data state while the simulation maintained high impedance.

Illegal Instruction Exception

The Illegal instruction (4AFC) allows the user to force an illegal instruction exception sequence. During this sequence, the status register is copied, the supervisor state entered, and the trace state turned off. A vector number is generated to refer to the illegal instruction vector. The current program counter (address of illegal instruction) and a copy of the status register are saved on the supervisor stack. Processing resumes at the address contained in the exception vector (11:66).

During the execution of a MC68000 instruction, the program counter points to the next instruction so that it is

in position for the current instruction's prefetch cycle. Therefore, during the model transformation process, the CDL microstatement "PC <- PCsub4" at phase two of clock cycle one was changed to "PC = PC - 2" in its ISP' counterpart in order to accurately locate the address of the illegal instruction.

For this instruction, address lines A1-A4 were monitored along with the lower eight data bus lines D0-D7 to facilitate data bus analysis. The simulation results were:

```
                 19-16 15-12 11-8 7--4 3--0 (columns)
                  AAAA  DDDD DDDD AULR DFFF (signals)
                  4321  7654 3210 SDD/ TCCC
                                   SSW A210
                                       C
                                       K
                 -------------------------
 K(0) P(1)        1111  1111 1111 1111 1010 $K(0) thru K(4) is a
 K(0) P(2)        0110  1111 1111 1111 1010 $read cycle just like
                                            $the one described
 K(1) P(1)        0110  1111 1111 0001 1010 $for the MOVE.W Dn,Dn
 K(1) P(2)        0110  1111 1111 0001 1010 $instruction.

 K(2) P(1)        0110  1111 1111 0001 1010
 K(2) P(2)        0110  1111 1111 0001 1010

 K(3) P(1)        0110  0000 0001 0001 0010 $Data (8-15) is
 K(3) P(2)        0110  0000 0001 0001 0010 $code for
                                            $for MOVE.W D1,D2.
 K(4) P(1)        0110  0000 0001 0001 0010
 K(4) P(2)        0110  0000 0001 1111 1010 $End read cycle

 K(5) P(1)        0110  1111 1111 1111 1010 $Data Bus (8-15)
 K(5) P(2)        1111  1111 1111 1111 1010 $and address bus
                                            $(16-19) go high.
 K(6) P(1)        1111  1111 1111 1111 1010
 K(6) P(2)        1111  1111 1111 1111 1010

 K(7) P(1)        1111  1111 1111 1111 1010
 K(7) P(2)        1111  1111 1111 1111 1010

 K(8) P(1)        1111  1111 1111 1111 1010
 K(8) P(2)        1111  1111 1111 1111 1010
```

```
 K(9) P(1)   1111   1111 1111 1111 1010 $Begin a write cycle
 K(9) P(2)   0010   1111 1111 1111 1101 $just like the one
                                         $described for the
K(10) P(1)   0010   1111 1111 0110 1101 $MOVE.W Dn,(An)
                                         $instruction, except
                                         $for function codes
                                         $and data. Function
                                         $code (0-2) is
                                         $supervisor data
                                         $mode.
K(10) P(2)   0010   0000 1010 0110 1101 $Data (8-15) is low
                                         $word of the PC.
K(11) P(1)   0010   0000 1010 0000 1101
K(11) P(2)   0010   0000 1010 0000 1101

K(12) P(1)   0010   0000 1010 0000 1101
K(11) P(2)   0010   0000 1010 0000 1101

K(12) P(1)   0010   0000 1010 0000 0101
K(12) P(2)   0010   0000 1010 0000 0101

K(13) P(1)   0010   0000 1010 0000 0101
K(13) P(2)   0010   0000 1010 1110 1101 $End Write Cycle.

K(14) P(1)   1111   0000 1010 1111 1101 $Begin write cycle
K(14) P(2)   0000   1111 1111 1111 1101 $just like preceding
                                         $one except for data.
K(15) P(1)   0000   1111 1111 0110 1101
K(15) P(2)   0000   0000 0100 0110 1101 $Data (8-15) is SR
                                         $contents.
K(16) P(1)   0000   0000  0100 0000 1101
K(16) P(2)   0000   0000  0100 0000 1101

K(17) P(1)   0000   0000  0100 0000 1101
K(16) P(2)   0000   0000  0100 0000 1101

K(17) P(1)   0000   0000  0100 0000 0101
K(17) P(2)   0000   0000  0100 0000 0101

K(18) P(1)   0000   0000  0100 0000 0101
K(18) P(2)   0000   0000  0100 1110 1101 $End write cycle.

K(19) P(1)   1111   0000  0100 1111 1101 $Begin write cycle
K(19) P(2)   0001   1111  1111 1111 1101 $just like preceding
                                         $one except for data.
K(20) P(1)   0001   1111  1111 0110 1101
K(20) P(2)   0001   0000  0000 0110 1101 $Data (8-15) is high
                                         $word of PC.
K(21) P(1)   0001   0000  0000 0000 1101
K(21) P(2)   0001   0000  0000 0000 1101

K(22) P(1)   0001   0000  0000 0000 1101
K(21) P(2)   0001   0000  0000 0000 1101

K(22) P(1)   0001   0000  0000 0000 0101
```

```
K(22) P(2)  0001  0000  0000 0000 0101

K(23) P(1)  0001  0000  0000 0000 0101
K(23) P(2)  0001  0000  0000 1110 1101 $End write cycle.

K(24) P(1)  1111  0000  0000 1111 1101 $Begin read cycle
K(24) P(2)  1000  1111  1111 1111 1101 $just like above
                                       $except data and
K(25) P(1)  1000  1111  1111 0001 1101 $function codes.
K(25) P(2)  1000  1111  1111 0001 1101

K(26) P(1)  1000  1111  1111 0001 1101
K(26) P(2)  1000  1111  1111 0001 1101

K(27) P(1)  1000  0000  0000 0001 0101 $Data (8-15) is high
K(27) P(2)  1000  0000  0000 0001 0101 $word of the address
                                       $of the exception
K(28) P(1)  1000  0000  0000 0001 0101 $handler routine.
K(28) P(2)  1000  0000  0000 1111 1101 $End read cycle.

K(29) P(1)  1111  1111  1111 1111 1101 $Begin read cycle
K(29) P(2)  1001  1111  1111 1111 1101 $just like
                                       $preceding one
K(30) P(1)  1001  1111  1111 0001 1101 $except for data.
K(30) P(2)  1001  1111  1111 0001 1101

K(31) P(1)  1001  1111  1111 0001 1101
K(31) P(2)  1001  1111  1111 0001 1101

K(32) P(1)  1001  0000  0010 0001 0101 $Data (8-15) is low
K(32) P(2)  1001  0000  0010 0001 0101 $word of address of
                                       $exception handler
K(33) P(1)  1001  0000  0010 0001 0101 $routine.
K(33) P(2)  1001  0000  0010 1111 1101 $End read cycle.

K(34) P(1)  1001  1111  1111 1111 1101 $K(34) thru K(38) is
K(34) P(2)  0001  1111  1111 1111 1110 $a read cycle just
                                       $like preceding one
K(35) P(1)  0001  1111  1111 0001 1110 $except data and
                                       $function codes.
K(35) P(2)  0001  1111  1111 0001 1110 $Function code (0-2)
                                       $supervisor program
K(36) P(1)  0001  1111  1111 0001 1110 $mode.
K(36) P(2)  0001  1111  1111 0001 1110

K(37) P(1)  0001  0111  0011 0001 0110 $Data (8-15) is
K(37) P(2)  0001  0111  0011 0001 0110 $return from
                                       $exception used as
K(38) P(1)  0001  0111  0011 0001 0110 $exception handler
                                       $routine
K(38) P(2)  0001  0111  0011 0001 0110 $End read cycle.

K(39) P(1)  1111  1111  1111 1111 1110 $Data bus (8-15)
K(39) P(2)  1111  1111  1111 1111 1110 $and address bus
                                       $go high.
```

```
K(40) P(1)  1111  1111  1111 1111 1110
K(40) P(2)  1111  1111  1111 1111 1110 $End exception
                                       $processing
```

Per the introductory discussion regarding the high impedance state of the address bus, at phase one of any new cycle, differences occur on ABUS whenever it is changing to the high impedance state. These take place at phase one of clock cycles 5, 14, 19, 24, 29, 34, and 39. Contrary to the earlier discussion regarding the high impedance state of the data bus, there are no differences on the data bus at phase one of those cycles that follow a write cycle. The logic analyzer's inability to capture its true state was anticipated and the instruction was modeled so that the data bus appears to return to the high impedance state during phase two.

Because the contents of the system stack pointer is repeatedly decremented and placed on the address bus during this sequence, its initial contents should be known in order to accurately describe the value on the data bus at all times. The ECB initializes the system stack pointer (SYSTACK) with the value 0786.

Address Error Exception

An address exeception sequence is initiated whenever the 68000 attempts to fetch a word or longword operand from an odd address. Whenever this occurs, the current bus cycle is aborted, the processor terminates current processing and begins a 60-clock cycle exception sequence that includes four read and seven write cycles.

The odd address error was generated by attempting to move a word from data register D1 to an odd memory location with the instruction "MOVE.W D1,(A1)" (A1 was initialized to 2001). The address bus is not monitored during the execution of this instruction. The simulation results were:

```
                  15-12 11-8 7--4 3--0 (columns)
                   DDDD DDDD AULR DFFF (signals)
                   7654 3210 SDD/ TCCC
                             SSW A210
                                 C
                                 K
                  --------------------
K(0) P(1)   1111 1111 1111 1010 $Begin read cycle.
K(0) P(2)   1111 1111 1111 1010

K(1) P(1)   1111 1111 0001 1010
K(1) P(2)   1111 1111 0001 1010

K(2) P(1)   1111 1111 0001 1010
K(2) P(2)   1111 1111 0001 1010

K(3) P(1)   0111 0001 0001 0010 $Data applied to data bus
K(3) P(2)   0111 0001 0001 0010 $(8-15). Data is code for
                                $NOP, so this is a
K(4) P(1)   0111 0001 0001 0010 $prefetch.
K(4) P(2)   0111 0001 1111 1010 $End read cycle.

K(5) P(1)   1111 1111 1111 1010 $Begin a write cycle.
K(5) P(2)   1111 1111 1111 1001

K(6) P(1)   1111 1111 0110 1001
K(6) P(2)   0101 0101 0110 1001 $Data put on data bus
                                $(8-15). data is 55 hex,
K(7) P(1)   0101 0101 0000 1001 $data being moved.
K(7) P(2)   0101 0101 0000 1001

K(8) P(1)   0101 0101 0000 1001
K(8) P(2)   0101 0101 1110 1001

K(0) P(1)   0101 0101 1111 1001 $Write cycle terminated
K(0) P(2)   1111 1111 1111 1001 $because of address error.
                                $Data bus (8-15) goes
K(1) P(1)   1111 1111 1111 1001 $high.
K(1) P(2)   1111 1111 1111 1001

K(2) P(1)   1111 1111 1111 1001
K(2) P(2)   1111 1111 1111 1001
```

```
K(3) P(1)   1111 1111 1111 1001
K(3) P(2)   1111 1111 1111 1001

K(4) P(1)   1111 1111 1111 1001
K(4) P(2)   1111 1111 1111 1001

K(5) P(1)   1111 1111 1111 1001
K(5) P(2)   1111 1111 1111 1001

K(6) P(1)   1111 1111 1111 1001
K(6) P(2)   1111 1111 1111 1001

K(7) P(1)   1111 1111 1111 1001
K(7) P(2)   1111 1111 1111 1001

K(8) P(1)   1111 1111 1111 1001 $Begin write cycle
K(8) P(2)   1111 1111 1111 1101 $Function codes (0-2)
                                $change to supervisor
K(9) P(1)   1111 1111 0110 1101 $data mode.
K(9) P(2)   0000 1000 0110 1101 $Data put on data bus
                                $(8-15). Data is low
K(10) P(1)  0000 1000 0000 1101 $word of PC.
K(10) P(2)  0000 1000 0000 1101

K(11) P(1)  0000 1000 0000 1101
K(10) P(2)  0000 1000 0000 1101

K(11) P(1)  0000 1000 0000 0101
K(11) P(2)  0000 1000 0000 0101

K(12) P(1)  0000 1000 0000 0101
K(12) P(2)  0000 1000 1110 1101 $End write cycle.

K(13) P(1)  0000 1000 1111 1101 $Begin write cycle.
K(13) P(2)  1111 1111 1111 1101

K(14) P(1)  1111 1111 0110 1101
K(14) P(2)  0000 0000 0110 1101 $Data put on data bus
                                $(8-15). Data is SR
K(15) P(1)  0000 0000 0000 1101 $contents.
K(15) P(2)  0000 0000 0000 1101

K(16) P(1)  0000 0000 0000 1101
K(15) P(2)  0000 0000 0000 1101

K(16) P(1)  0000 0000 0000 0101
K(16) P(2)  0000 0000 0000 0101

K(17) P(1)  0000 0000 0000 0101
K(17) P(2)  0000 0000 1110 1101 $End write cycle.

K(18) P(1)  0000 0000 1111 1101 $Begin write cycle.
K(18) P(2)  1111 1111 1111 1101
```

```
K(19) P(1)  1111 1111 0110 1101
K(19) P(2)  0000 0000 0110 1101 $Data put on data bus
                                $(8-15). Data is high
K(20) P(1)  0000 0000 0000 1101 $word of PC.
K(20) P(2)  0000 0000 0000 1101

K(21) P(1)  0000 0000 0000 1101
K(20) P(2)  0000 0000 0000 1101

K(21) P(1)  0000 0000 0000 0101
K(21) P(2)  0000 0000 0000 0101

K(22) P(1)  0000 0000 0000 0101
K(22) P(2)  0000 0000 1110 1101 $End write cycle.

K(23) P(1)  0000 0000 1111 1101 $Begin write cycle
K(23) P(2)  1111 1111 1111 1101

K(24) P(1)  1111 1111 0110 1101
K(24) P(2)  1000 0001 0110 1101 $Data put on data bus
                                $(8-15). Data is code
K(25) P(1)  1000 0001 0000 1101 $for instruction being
K(25) P(2)  1000 0001 0000 1101 $executed when interrupt
                                $occurred.
K(26) P(1)  1000 0001 0000 1101
K(25) P(2)  1000 0001 0000 1101

K(26) P(1)  1000 0001 0000 0101
K(26) P(2)  1000 0001 0000 0101

K(27) P(1)  1000 0001 0000 0101
K(27) P(2)  1000 0001 1110 1101 $End write cycle.

K(28) P(1)  1000 0001 1111 1010 $Begin a write cycle.
K(28) P(2)  1111 1111 1111 1101

K(29) P(1)  1111 1111 0110 1101
K(29) P(2)  0000 0001 0110 1101 $Data put on data bus
                                $(8-15). Data is low
K(30) P(1)  0000 0001 0000 1101 $word of memory being
K(30) P(2)  0000 0001 0000 1101 $used when interrupt
                                $occurred.
K(31) P(1)  0000 0001 0000 1101
K(30) P(2)  0000 0001 0000 1101

K(31) P(1)  0000 0001 0000 0101
K(31) P(2)  0000 0001 0000 0101

K(32) P(1)  0000 0001 0000 0101
K(32) P(2)  0000 0001 1110 1101 $End Write Cycle.

K(33) P(1)  0000 0001 1111 1101 $Begin write cycle.
K(33) P(2)  1111 1111 1111 1101

K(34) P(1)  1111 1111 0110 1101
```

```
K(34) P(2)  0000 0001 0110 1101 $Data put on data bus
                                $(8-15). Data is cycle
K(35) P(1)  0000 0001 0000 1101 $type (R/W) and function
K(35) P(2)  0000 0001 0000 1101 $codes when interrupt
                                $occurred.
K(36) P(1)  0000 0001 0000 1101
K(35) P(2)  0000 0001 0000 1101

K(36) P(1)  0000 0001 0000 0101
K(36) P(2)  0000 0001 0000 0101

K(37) P(1)  0000 0001 0000 0101
K(37) P(2)  0000 0001 1110 1101 $End write cycle.

K(38) P(1)  0000 0001 1111 1101 $Begin write cycle.
K(38) P(2)  1111 1111 1111 1101

K(39) P(1)  1111 1111 0110 1101
K(39) P(2)  0000 0000 0110 1101 $Data put on data bus
                                $(8-15). Data is high
K(40) P(1)  0000 0000 0000 1101 $word of memory being
K(40) P(2)  0000 0000 0000 1101 $used when interrupt
                                $occurred.
K(41) P(1)  0000 0000 0000 1101
K(40) P(2)  0000 0000 0000 1101

K(41) P(1)  0000 0000 0000 0101
K(41) P(2)  0000 0000 0000 0101

K(42) P(1)  0000 0000 0000 0101
K(42) P(2)  0000 0000 1110 1101 $End write cycle.

K(43) P(1)  0000 0000 1111 1101 $Begin read cycle.
K(43) P(2)  1111 1111 1111 1101

K(44) P(1)  1111 1111 0001 1101
K(44) P(2)  1111 1111 0001 1101

K(45) P(1)  1111 1111 0001 1101
K(45) P(2)  1111 1111 0001 1101

K(46) P(1)  0000 0000 0001 0101 $Data applied to data bus
K(46) P(2)  0000 0000 0001 0101 $(8-15). Data is high
                                $word of address of
K(47) P(1)  0000 0000 0001 0101 $exception handler routine.
K(47) P(2)  0000 0000 1111 1101 $End read cycle.

K(48) P(1)  1111 1111 1111 1101 $Begin read cycle.
K(48) P(2)  1111 1111 1111 1101

K(49) P(1)  1111 1111 0001 1101
K(49) P(2)  1111 1111 0001 1101

K(50) P(1)  1111 1111 0001 1101
K(50) P(2)  1111 1111 0001 1101
```

```
K(51) P(1)  0100 0000 0001 0101 $Data applied to data bus
K(51) P(2)  0100 0000 0001 0101 $(8-15). Data is low
                                $word of address of
K(52) P(1)  0100 0000 0001 0101 $exception handler routine.
K(52) P(2)  0100 0000 1111 1101 $End read cycle.

K(53) P(1)  1111 1111 1111 1101 $Begin read cycle.
K(53) P(2)  1111 1111 1111 1110 $Function codes (0-2)
                                $change to supervisor
K(54) P(1)  1111 1111 0001 1110 $program mode.
K(54) P(2)  1111 1111 0001 1110

K(55) P(1)  1111 1111 0001 1110
K(55) P(2)  1111 1111 0001 1110

K(56) P(1)  1000 1111 0001 0110 $Data applied to data bus
K(56) P(2)  1000 1111 0001 0110 $(8-15). Data is code for
                                $first instruction of
K(57) P(1)  1000 1111 0001 0110 $exception handler routine.
K(57) P(2)  1000 1111 1111 1110 $End read cycle.

K(58) P(1)  1111 1111 1111 1110 $Data bus goes high.
K(58) P(2)  1111 1111 1111 1110

K(59) P(1)  1111 1111 1111 1110
K(59) P(2)  1111 1111 1111 1110 $End exception
                                $processing
```

The initial difference between the simulation and logic analyzer results begins at phase two of the second clock cycle 15 (wait state) and continues through phase two of clock cycle 17. During this time period, the logic analyzer data shows the data bus at state "0001 0000" while the simulation indicates state "0000 0000". This is the write cycle that saves the contents of the status register at the time the illegal address reference was made. The state represented by the logic analyzer data is inaccurate.

The state of the status register at the time of the address error was zero because only the supervisor mode bit (bit 13) was addressed during the test routine, and it was set to zero by the "AND.W #$DFFF,SR" instruction to place

the 68000 in the user mode. The four condition code bits, interrupt mask, or trace bit were not altered from their original low states. The SR is placed on the data bus at phase two of clock cycle 14 and should remain there for the duration of the write cycle (phase two of clock cycle 17). This information, along with the fact that the logic analyzer result could not be duplicated, suggests that this is a consistent typing error.

Another difference begins at phase two of clock cycle 34 and continues through phase 2 of clock cycle 37. Here the logic analyzer data depicts the data bus at state "1000 0001" while the simulation indicates "0000 0001". This data represents the type of memory access that was attempted at the time of the exception: information such as whether it was a read or write, whether the processor was processing an instruction or not, and the state of the function code outputs when the address error occured (11:67). The format of the access type is depicted in Figure VI-4.

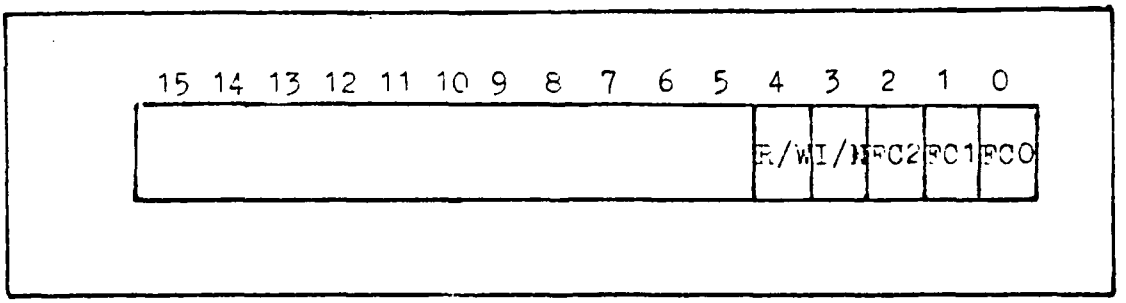


Figure VI-4. Memory Access Type (11:68)

Neither Figure VI-4 nor any of the other available documentation indicate that bit seven is used by the 68000. Because this analyzer output was duplicated, it appears this

portion of the address error exeception cannot be accurately modeled without additional documentation.

The format of the memory access type also revealed an error in the CDL model for this instruction. At phase one of clock cycle eight, the microstatement "ACTYPE(3) <- EXCEPT" was changed to "ACTYPE<3> = lo" in the ISP' version. This bit indicates whether the 68000 was processing an instruction at the time of the exception or not (instruction = 0, not = 1) (11:68). Because EXCEPT was in a high state, assigning it to ACTYPE(3) would incorrectly indicate that the 68000 was not processing an instruction at the time of the exception.

At phase two of clock cycle 23, this instruction's CDL model contains the microstatement "IDBUS <- IR". The objective is to save the instruction causing the address exception (11:67). However, since the instruction that erred was placed in a temporary instruction register at phase two of clock cycle seven with the microstatement "IRTEMP <- IR", and then IR was reused at phase two of clock cycle eight, the saving microstatement will not achieve the desired results. It was instead replaced with the microstatement "IDBUS = IRTEMP" in the ISP' version.

The simulation's data bus also differs from the logic analyzer's results at phase two of clock cycles 45 and 55. The simulation indicates a high impedance state (1111 1111) and logic analyzer data shows "1011 1111". Similar to the simulation of the MOVE.W D1, 04(A1) instruction, the logic

analyzer appears to have caught the data bus in transition because the differences occur when the memory is placing data on the bus during read cycles. The ISP' model for this instruction also anticipated the delay in showing the data bus at the high impedance state following write cycles.

VII. Conclusions and Recommendations

Introduction

This thesis was a follow-on effort to another project whose objective was to develop a functional model of the Motorola MC68000 that was capable of being simulated. The model was to be constructed through signal analysis supported only by technical data normally provided to a retail purchaser of the system (8:I-6,7). The natural objective of this succeeding project was to simulate the models that were developed using that approach, analyze the results, and then draw conclusions regarding its viability based on the accuracy and completeness of the models. This chapter will present the conclusions resulting from this study, as well as suggest further areas of research that may broaden the conclusions reached regarding this innovative method of microprocessor modeling and simulation.

Conclusions

The completeness and accuracy of the models developed by Hamby and Guillory demonstrate that the MC68000 can be modeled at the functional level through signal analysis supported by technical literature available to the public. The state of the MC68000 was accurately depicted at each clock cycle when executing a variety of instructions and exception sequences. Only the dynamics of the address and data buses during transitional periods prevent the state of

each of the MC68000's signals from being totally deterministic for every clock cycle. Even when the state of the bus is otherwise deterministic, the lack of sophistication of the monitoring equipment sometimes prevented it from being accurately captured. But because these indeterminate states occur at cycles which are of no functional consequence to correct processor operation, they can be ignored in terms of their impact on the accuracy of the model.

Based on the mix of instructions, exceptions, and addressing modes modeled, it appears that there would be no difficulty in modeling the complete set of MC68000 instructions and exception processing states. This conclusion can reasonably be extended to include other processors whose architectural complexity does not exceed the MC68000's. However, no definite conclusions can be reached regarding the effectiveness of this modeling approach on other processors with more complex architectures or sophisticated implementation techniques. Even some architectures less sophisticated than the MC68000 may prove more troublesome to model with this approach. For example, the 16-bit Intel 8088 multiplexes a subset of its 20 address lines to support eight-bit data transfers. This may hamper this modeling approach by making the process of capturing, distinguishing, and interpreting the address and data signals much more difficult.

Some of the more sophisticated 32-bit architectures

employing extensive parallelism and pipelining may prove to be even much more challenging. A good example is the latest version of the MC68000 family of microprocessors, the MC68020. The MC68020 maintains a 256-byte-on-chip instruction cache. This capability removes the instruction stream from the data bus making instruction identification more difficult as they are executing. This cache implementation also allows simultaneous instruction and data accesses to occur hiding the execution of some microinstructions from the observer. The technical data necessary to accurately model the processor's management of the instruction cache may also be difficult to obtain. The algorithms employed to initialize the cache, update both it and main memory during writes, and reload the cache during misses would be difficult to discern from signal analysis alone. Further increasing the difficulty of employing this modeling approach on the MC68020 is the fact that the instruction pipeline has been increased from two to three words thereby allowing three instructions to concurrently undergo the process of decoding and execution.

Another processor with an imposing architecture that may be extremely difficult to model with this approach is Intel's iAPX 432. The iAPX 432 is actually a three-chip set with each packaged in a 64-pin quad in-line package (QUIP). Two of the chips combine to form the General Data Processor (GDP): the iAPX 43201 (instruction decode unit) and the iAPX 43202 (instruction execution unit). These two chips

communicate across a microinstruction bus to fetch, decode, and execute program instructions. A third chip, the iAPX 43203 interface processor coordinates all I/O between peripherals and memory. An attached processor such as the Intel 8086 is used in conjunction with the iAPX 43203 to form an I/O processor unit for the iAPX 432 system. The iAPX 43203 communicates with the attached processor via a subsystem bus and with the GDP using a "packet" bus. The packet bus uses data packets that vary from one to sixteen bytes in length to provide communications between the GDP, memory, and interface processor. The number of chips, buses, and signals comprising the iAPX 432 system, coupled with the packeting of bus information would certainly make the task of modeling the iAPX using the approach under study very difficult, if not impossible.

The complexity of the instruction set also becomes an issue when considering a high-level-language architecture such as the iAPX 432. The iAPX 432 has 230 instructions with lengths that vary from six to 344 bits. Each of the instruction fields are also bit-variable and can encase up to six operands. Instructions such as "CREATE-DATA-SEGMENT" may consist of hundreds of microoperations requiring a significant number of clock cycles to execute. The identification of both macro and microinstructions embedded in packets transmitted over multiple buses for an extended number of clock cycles will require extensive and detailed technical data supported by several sophisticated monitoring

devices.

A point that should be made is that even though it may be extremely difficult to gather the data necessary to model a system such as the iAPX 432 using this approach, there certainly would be no difficulty in coding and simulating the model using N.mPc. One of N.mPc's strong suits is its ability to describe and simulate networked multiple microprocessor systems such as the iAPX 432. Each of the iAPX 432's three processors, its supporting memory, and the attached processor could be independently modeled and linked through a description of their multiple connecting bus structures to form a simulatable system from which operational performance data could be gathered.

Recommendations

1. Model other microprocessors using this approach to determine which architectural implementations lend themselves to this modeling method. Architectural designs or implementation techniques forming barriers to this approach could then be identified and documented. An upper bound in terms of architectural complexity that can be modeled with confidence using this approach could be established. Some candidates in order of their increasing complexity are the aforementioned microprocessors; the Intel 8088, the MC68020, and the Intel iAPX 432.

2. Develop a generalized and optimized model of the MC68000 with N.mPc. By conforming with the CDL originals, the current ISP' models became too primitive and

specialized. Each instruction was independently modeled to process specific operands for a particular addressing mode. Each contained its own internal memory and timing signals. Not only does such an arrangement not represent the real world environment, but to model the MC68000's complete set of instructions in all of their addressing modes would have required over 1000 models averaging 50 kbytes in length each. This number would escalate greatly if each instruction were not assumed to receive a generalized operand set. A complete model of the MC68000 that consolidates instruction models of this type is obviously out of the question. A more practical approach is to discard the previous model structure and adopt a more general composition that will enable the researcher to take advantage of the capabilities of a nonprecedural language such as ISP' and its host system, N.mPc.

Generalized routines to model multiple instructions in each of their addressing modes and accept any of their legitimate operands could easily be developed to minimize the model size while maximizing its ability to completely describe the MC68000. For example, the fourteen models developed during this project to represent variations of the "MOVE" instruction could be consolidated into a single procedure that received its operands, the data size, and its addressing modes as parameters. It would itself make calls to general lower-level routines as necessary to accomplish the microinstructions associated with a particular

variation. The memory and two-phase clocking signals could be modeled as independent system entities that communicate with the MC68000 over a system bus. Such a representation would afford an opportunity to model, simulate, and analyze the MC68000 at a level of detail not heretofore achieved.

VIII. Analysis of Time Spent on Project

Introduction

One of the requirements of this thesis was to maintain a detailed log of all time spent on the project. This chapter presents an analysis of that time log.

Summary of Time

Project time was spent as follows:

1. Background Research and Project Preparation — 18 hrs
   - Identifying Research Objectives
   - Performing Literature Search
   - Reviewing Texts and Articles
   - Studying Predecessor Thesis
   - Formulating Research Approach
2. N.mPc Installation and Familiarization — 91 hrs
   - Installing, Configuring, and Testing System
   - Generating System Documentation
   - N.mPc Familiarization
3. MC68000, ECB, and Logic Analyzer Familiarization — 25 hrs
   - Reviewing Texts, Articles, and Technical Documentation
   - Programming ECB
   - Operating Logic Analyzer
4. CDL-to-ISP' Model Transformations — 58 hrs
   - CDL and ISP' Familiarization
   - Formulating Transformation Methodology

- Coding ISP' Models

5. Simulate Models 60 hrs
   - Develop Simulation Strategy
   - Building Individual Instruction Simulations
   - Running Simulations
   - Testing and Debugging Simulations
   - Generating Simulation Hardcopies
6. Simulation Analysis 42 hrs
   - Reviewing Simulation Data
   - Reviewing Logic Analyzer Data
   - Re-examining MC68000 Operation
   - Correcting Model Discrepancies
7. Thesis Preparation 418 hrs
   - Writing
   - Typing
   - Editing
   - Printing

Hamby and Guillory speculated in their thesis that it would require a team of two people approximately six months to test and model the entire MC68000 processor using this approach (8:VII-2,3). I further recommend that one of the team members be "software-oriented" to facilitate the development of a generalized model as recommended in Chapter VII. Such a model will require someone experienced in the areas of software engineering, modular programming, etc. Accepting their estimate, this team configuration will allow the development and simulation of a generalized and

optimized model within the same time frame.

Bibliography

1. ADI-814-I. MC68000 Advance Information. Austin, Texas: Motorola Semiconductor Products Inc., 1982.

2. Barbacci, Mario R. "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems," IEEE Transactions on Computers, C-24 (2): 137-148 (February 1975).

3. Bryce, Heather. "Microprogramming Makes the MC68000 a Processor Ready for the Future," Electronic Design (October 25, 1979).

4. Chu, Yaohan. Computer Organization and Microprogramming. Englewood Cliffs, N.J.: Prentice-Hall Inc., 1972.

5. Drongowski, Paul J. "Functional Simulation with the N.mPc System," VLSI DESIGN: 76-77 (Jan 1984).

6. Drongowski, Paul J. et al. "A Guide for Writing N.mPc Hardware Models." Department of Computer Engineering and Science, Case Western Reserve University, Cleveland OH, January 1984.

7. Fong, James Y. O. "Microprocessor Modeling for Logic Simulation," Proceedings of the IEEE 1981 International Test Conference: 458-460 (October 1981).

8. Hamby, James R. and Galen J. Guillory. Architectural Analysis and Modeling of a Motorola MC68000 Microprocessor. MS Thesis, GE/EE/83D-23. School of Engineering, Air Force Institute of Technology (AU), Wright Patterson AFB, Ohio, December 1983.

9. Hewitt, Donald C. Jr. The Runtime Environment for N.mPc, An Adaptable Software System to Support the Development of Microprocessor-Based Systems. MS Thesis CES-79-7. Department of Computer Engineering and Science, Case Western Reserve University, Cleveland OH, January 1979.

10. MC68000UM[AD2]. MC68000 16-Bit Microprocessor User's Manual (Second Edition). N.J.: Prentice-Hall, 1982.

11. MC68000UM[AD3]. MC68000 16-Bit Microprocessor User's Manual (Third Edition). N.J.: Prentice-Hall, 1982.

12. NP-355-R1. MC68000 Product Review. Austin, Texas: Motorola Semiconductor Products Inc., 1982.

13. Ordy, Greg M. N.mPc: Ecologist User's Manual. Department of Computer Engineering and Science, Case Western Reserve University, Cleveland OH, March 1979.

14. Ordy, Greg M. N.mPc: Runtime User's Manual. Department of Computer Engineering and Science, Case Western Reserve University, Cleveland OH, March 1979.

15. RADC-TR-81-343. Product Evaluation Report of the Motorola MC68000. Griffis AFB, N.Y.: Rome Air Development Center, Nov 1981.

16. Rogers, Lawrence R. and Greg M. Ordy. The Metamicro User's Manual, Version 3.1. Department of Computer Engineering and Science, Case Western Reserve University, Cleveland OH, March 1980.

17. Rogers, Lawrence R. The Linking/Loader User's Manual, Version 1.1. Department of Computer Engineering and Science, Case Western Reserve University, Cleveland OH, March 1980.

18. Rose, Charles W. et al. "N.mPc: A Study in University-Industry Technology Transfer," IEEE Design and Test: 44-56 (Feb 1984).

19. Shah, Samir S. An Assessment of N.mPc for 16-Bit Systems. MS Thesis CES-80-10. Department of Computer Engineering and Science, Case Western Reserve University, Cleveland OH, August 1980.

20. Straubs, R. V. ISP' User's Manual. MS Thesis CES-78-8. Department of Computer Engineering and Science, Case Western Reserve University, Cleveland OH, August 1978.

21. Stritter, Edward and Tom Gunter. "A Microprocessor Architecture for a Changing World: The Motorola 68000," Microsystems (February 1979).

22. Titus, Christopher A. et al. 16-Bit Microprocessors. Indianapolis: Howard W. Sams & Co., 1982.

23. Toong, Hoo-min D. and Amar Gupta. "An Architectural Comparison of Contemporary 16-Bit Microprocessors", IEEE Transactions on Computers, (May 1981).

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

AD-A164 203

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/ENG/84-D-2	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable): AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFSC/FTD; 8b. OFFICE SYMBOL (If applicable): TQTA	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. — PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification): See Box 19	
12. PERSONAL AUTHOR(S): Charles A. Baxley Jr., B.S., Capt, USAF	
13a. TYPE OF REPORT: MS Thesis; 13b. TIME COVERED: FROM _____ TO _____; 14. DATE OF REPORT (Yr., Mo., Day): 1984 December; 15. PAGE COUNT: 1200	
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB GR	Computer Design Language(CDL), Instruction Set Processor (ISP'), N.mPc(Networked Microprocessor) Motorola MC68000, Microprocessor Modeling,
09	02		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: THE SIMULATION AND ANALYSIS OF A RTL MODEL OF THE MOTOROLA MC68000 MICROPROCESSOR WITH N.mPc

Thesis Chairman: Frederick A. Zapka, Major, USA

Approved for public release: IAW AFR 190-17.
LYNN E. WOLAVER 14 AUG 85
Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED [X] SAME AS RPT [ ] DTIC USERS [ ]	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Frederick A. Zapka, Major, USA	22b. TELEPHONE NUMBER (Include Area Code): 513 255 3054; 22c. OFFICE SYMBOL: AFIT/ENG

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE

18. Microprocessor Simulation, Microprocessor Analysis, Computer Architecture, Microcomputers, Computerized Simulation, Digital Simulation.

19. In a prior thesis project, a functional level model of portions of the Motorola MC68000 microprocessor was developed using signal analysis supported by limited technical data. Representative parts of the instruction set and exception processing structure were modeled with the Computer Design Language (CDL). In this follow-on effort, those CDL models are transformed into equivalent models using ISP', an enhanced vers on of the Instruction Set Processor (ISP) hardware design language. This language transformation enabled the models to be simulated using N.mPc, a VAX 11/780-hosted software package developed specifically to support the design of digital systems. To evaluate the correctness of the of the models, the simulation results are analyzed against signal data gathered with the aid of a logic analyzer during the actual operation of the MC68000 when processing the modeled instructions. The accuracy and completeness of the examined models suggests that this functional approach to microprocessor modeling is a valid one.

END

FILMED

4-86

DTIC