To the Graduate Council:

I am submitting herewith a thesis written by Danny F. Newport entitled "A Coprocessor VLSI Design Frame." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Donald W. Bouldin

Donald W. Bouldin, Major Professor

We have read this thesis
and recommend its acceptance:

Robert E. Barkhausen

Dwayne Bynum

Accepted for the Council:

Lew Minkel

Vice Provost
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Danny F. Nugent

Date November 26, 1986

A COPROCESSOR VLSI DESIGN FRAME

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Danny F. Newport

December 1986

DEDICATION

This thesis is dedicated to my wife, Rosemary, without whose love and support this work might never have been completed, and to the memory of my late grandfather, Rev. Daniel Daugherty, whose life and strength helped shape my life.

ACKNOWLEDGMENTS

The author wishes to express his appreciation to his major professor, Dr. Donald W. Bouldin, for his identification and guidance of the research. Special thanks go to Dr. R. E. Bodenheimer and Dr. D. Brzakovic, the members of his committee, for review of the thesis, and in particular, to Dr. Bodenheimer for his having asked, "Where's that thesis?" occasionally.

A special thanks is given to the author's employer, the College of Engineering at the University of Tennesse, and the Department of Electrical Engineering for their support and encouragement to obtain an advanced degree.

Finally, the author wishes to express his appreciation to his parents and family for their love, support, and encouragement throughout the years.

ABSTRACT

The purpose of this work was to design and specify the implementation of a coprocessor design frame for use in very large scale integrated (VLSI) circuit designs which would be pin compatible with an existing coprocessor.

Design frames are the interface between a custom integrated circuit and a standard bus or system. The design frame implements the necessary protocol to communicate with the bus or system while presenting the custom circuit with a much simpler protocol. This allows a designer to concentrate on the circuit design, not on the implementation of the communication protocol, and to install the fabricated design into an existing system for testing, evaluation, and use in actual production systems. A coprocessor design frame provides a custom VLSI circuit access to the tools and resources of virtually any microcomputer system which has a supported coprocessor interface.

This thesis presents a conceptual description of three design frames that implement a coprocessor interface to three widely available microprocessors – the Motorola 68020 and the Intel 8086/8088 and Intel 80286. These three conceptual descriptions were evaluated, and the coprocessor design frame for the Intel 8086/8088 was chosen for complete design and specification. By using this design frame, a custom VLSI design can be installed in any microcomputer system which is based on the Intel 8086/8088 and has an available coprocessor socket. Thus, a custom VLSI circuit can be quickly and easily used by a much larger audience.

TABLE OF CONTENTS

CHAPTER PAGE

CHAPTER PAGE

LIST OF FIGURES

FIGURE PAGE

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Advantages of Design Frames

As the ability to design and fabricate custom integrated circuits becomes available to more application designers, the need to incorporate a custom design into existing systems grows. Design frames, by providing an interface between a custom integrated circuit and a standard bus or system, facilitate this process.

In many cases, the task of designing the interface between a custom circuit and the outside world is comparable to that required to design the custom circuit. Design frames relieve the applications designer of this task by translating the protocol demanded by the bus or system into a much simpler protocol to be used by the custom circuit. Instead of having to interpret a large volume of technical literature, the designer need only follow a few simple guidelines permitting more concentration on the design of the custom circuit[7,8].

A design frame also implements the electrical and physical interfacing of the custom circuit to the outside world. All the circuitry required to interface to the bus or system is located on the periphery of the integrated circuit and a defined set of signal lines, including system clocking, are provided to the custom circuit. Connection pads and die size are provided to match standard pad frames available at fabrication services such as the MOS Implementation Service (MOSIS)[15].

By solving a number of problems that a designer faces, design frames offer significant benefits to a designer. Specifically, an idea for an integrated circuit can be rapidly prototyped and subsequently tested in an existing system. If design frames for different systems have the same internal protocol, a custom design need only be placed in the appropriate design frame to be used in a given system. As systems change, the custom design can be reused by being placed in a new design frame and fabricated. Since a design frame for a specific system needs to be designed only once, the effort of including a new interface each time a new circuit is designed is saved, and more than one designer can use the same design frame. Thus, design frames lead to a building-block approach of designing systems incorporating very large scale integrated (VLSI) circuits[6,8,9].

1.2 Existing Design Frames

Several design frames have been designed by others and are in use. The first design frame was devised by Alan Paeth, a researcher at the Xerox Palo Alto Research Center in California, and was essentially an interface to a set of LEDs and switches[13,14]. This "basic" design frame is used in many VLSI courses to provide students with a simple means to test their own fabricated designs with relative ease. A higher level design frame which is pin compatible with the Intel 8086 was developed in 1982[11]. This design frame implemented the bus controller interface to the 8086 processor bus, but does not implement the full 8086 logic (i.e. queue and queue status). A more sophisticated and systems-oriented design frame is the MULTIBUS[1] design frame developed in 1983[7].

[1]MULTIBUS is a trademark of Intel Corporation and is a patented Intel bus.

A custom integrated circuit designed to use the MULTIBUS design frame can easily access virtually any memory or device on the same MULTIBUS.

The only difficulty associated with any design frame is that it requires a specific external system. The "basic" design frame, as part of its specifications, requires a box with LEDs and switches. The Intel 8086 pin compatible design frame plugs into a socket on a single board computer which is then plugged into a MULTIBUS-based system where another processor can access the memory on the single board computer for communication and loading of code for the custom circuit. Similarly, the MULTIBUS design frame plugs into a socket on a specially designed printed circuit board which connects to a MULTIBUS-based system. Thus, design frames range from the simple to the complex, and many are waiting to be designed.

1.3 Scope of this Research

This thesis presents the conceptual description of three design frames that implement a coprocessor interface to three widely available microprocessors – the Motorola 68020 and the Intel 8086/8088 and 80286. These design frames would be pin compatible with an existing coprocessor for a specific microprocessor family. A coprocessor (called a processor extension by Intel) is typically a specialized processor which communicates and is connected directly with the main processor. For example, the Motorola 68881 is a numeric coprocessor for the 68020, whereas the Intel 8087 is a numeric coprocessor for the 8086/8088.

By implementing the same internal protocol in each of the coprocessor design frames, a custom integrated circuit can be used on whichever supported microprocessor system

is available. The internal protocol is kept very simple and is very similar to that used in the MULTIBUS design frame. The packaging of the design frame is selected to be identical to the current coprocessors available for a given microprocessor (i.e. a 68-pin grid-array for the Motorola 68881 and a 40-pin dual-inline package (DIP) for the Intel 8087). This allows the design frame and the custom circuit to be installed in the prewired coprocessor socket available on most microprocessor systems.

Although conceptual descriptions for all the design frames are presented, a complete design and specification is presented for only the coprocessor design frame for the Intel 8086/8088. This decision was based essentially on what microprocessor systems were available for an actual test and evaluation and what standard packages were available from the fabrication service, MOSIS. Intel 8086-based systems were available and easily accessible, and MOSIS provides a standard 40-pin DIP which matches the Intel 8087 requirement but not a 68-pin grid-array. MOSIS can provide the 68-pin grid-array but only as a special request with a much longer turnaround time. However, the fact that the coprocessor interface for the Intel 8086/8088 can gain more direct control of the system than the other interfaces did influence the decision.

1.4 Chapter Contents

The major design constraints and internal protocol definition for the general coprocessor design frame is presented in Chapter 2 along with compatibility with other design frames. Chapter 3 describes the process by which the decision was made to do a complete design for the Intel 8086/8088. A brief description of each microprocessor's coprocessor interface is also presented. The conceptual description of the Motorola 68020

coprocessor design frame is presented in Chapter 4 and Chapter 5 presents the conceptual description of the Intel 80286 coprocessor design frame. A brief description of the software requirements of each coprocessor design frame is also presented in these chapters. The complete design for the Intel 8086/8088 coprocessor design frame is presented in Chapter 6. Finally, Chapter 7 presents a summary and conclusions concerning the design and use of coprocessor design frames.

CHAPTER 2

COPROCESSOR DESIGN FRAME FUNDAMENTALS

2.1 Design Frame Constraints

Design frames provide a solution to the problem of integrating a custom VLSI design into an existing system. In the past, a circuit designer had to design both the custom circuit and the interface hardware. The interface hardware includes all of the necessary circuitry for the custom design to communicate with the outside world (i.e. input and output connection pads, latches, etc.) and in some cases requires as much effort to design as the custom design. But with a design frame, this design effort is not necessary. The design frame provides a much simpler protocol, both electrically and symbolically, for a custom design to be interfaced.

To provide an effective interface between a custom circuit and an existing system, a design frame should be designed based on *abstract* and *physical constraints.* There are five abstract constraints – conceptuality, universality, robustness, standardization, and systematics. The physical constraints are geometry, electrical specifications, and assembly[14].

A design frame should be conceptually easy for a designer to understand and use (conceptuality). At the same time, the design frame should be usable by a wide range of custom designs and be able to be manufactured on a variety of production lines (universality). The design should be sufficiently engineered such that the frame will not introduce any potential problems into the system (robustness). If a standard exists

for some specific aspect of the design frame, it should be used in the design frame (standardization). The naming, availability, and placement of cells and signals within the design frame should be systematic and uniform (systematics).

The layout and presentation of a design frame should be consistent and simple, thus allowing less developed design systems to use the design frame (geometry). For a custom circuit to use a design frame, the electrical characteristics of the design frame should be conservative and not use any exotic circuitry (electrical specifications). The packaging and installation of the design frame should be based on available packages and existing systems (assembly). By using these design constraints, both abstract and physical, a designer can develop a design frame which is not only simple and easy to use but functionally sound and usable by a large audience. A coprocessor interface complies with all of these design constraints.

2.2 General Coprocessor Overview

Most existing systems use a general purpose microprocessor to satisfy the needs of numerous computer applications. However, some specific applications require the use of specially designed hardware to give satisfactory levels of performance. This specially designed hardware can be implemented as a coprocessor to expand the capabilities of the microprocessor system. A coprocessor does not function as a standard peripheral on a microprocessor system, but as an extension to the microprocessor. Most currently available microprocessors,such as the Motorola 68020 and the Intel 8086/88 and 80286, have a predefined coprocessor interface. Except for specific implementation differences,

coprocessor interfaces are very similar. A typical microprocessor-coprocessor configuration is shown in Figure 2.1.

Coprocessors generally communicate to the main processor directly through some predetermined protocol and respond to instructions from the same instruction stream as the main processor. These instructions are different from those normally executed by the main processor, thus a coprocessor expands a system architecture. In a conventional peripheral, the programmer must implement the communication protocol between the peripheral and main processor, but in a coprocessor interface the communication protocol is handled by the hardware. Therefore, a coprocessor appears to the user as part of the main processor, not external hardware.

A design frame implementing a coprocessor interface forms an ideal platform for a VLSI application designer. The custom circuit would have access to the system through the main processor and, at the same time, could receive specific instructions to be executed. The operating system of the microprocessor system would not be affected by the addition of the custom circuit, which would appear as a coprocessor to the system. In many existing microprocessor systems, such as the IBM-PC/XT[1] and its many clones which are based on the Intel 8086/88, a prewired socket is provided for a coprocessor giving the designer a readily accessible system which could use the custom circuit (Figure 2.2). Thus, a coprocessor interface is suitable as a framework for a very versatile design frame.

[1]IBM-PC/XT is a trademark of the International Business Machines Corporation (IBM).

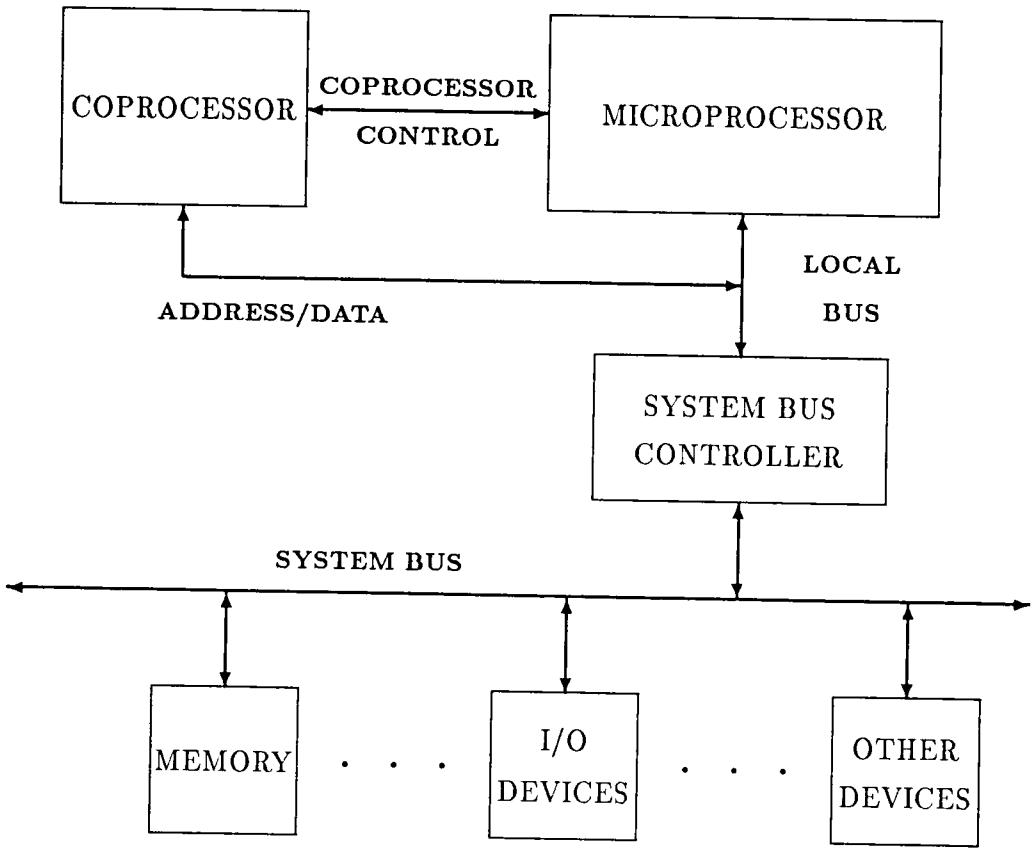


Figure 2.1: A typical microprocessor-coprocessor configuration.

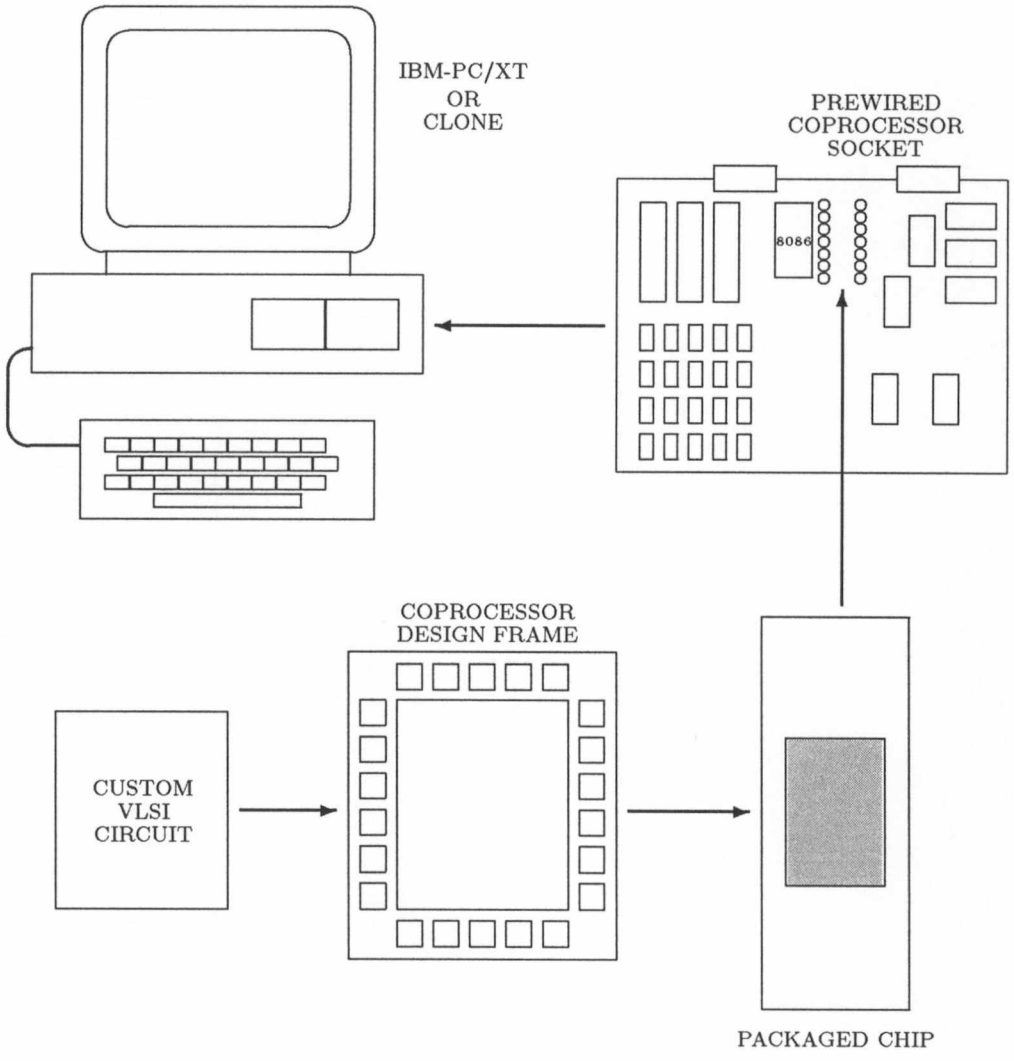


Figure 2.2: A coprocessor design frame for the Intel 8086 makes possible the use of custom VLSI circuits in an IBM-PC/XT or clone.

2.3 Internal Protocol Definition

For a design frame to achieve more widespread use and follow the main criteria of rapid prototyping and integration, it should support, as much possible, the same internal protocols as other design frames. In keeping with this philosophy, the internal protocol supported by the coprocessor design frames described in this thesis have been made as compatible as possible to the MULTIBUS design frame by Borriello, Katz, and Bell[7,8]. The coprocessor design frame for a specific microprocessor family has its own structure, but the internal protocols are the same.

The internal protocol of a design frame must fulfill a number of interface requirements for the custom circuit. These requirements include providing for clocking, communications, system addressing, data input and output, synchronizing logic, and signal availability. To maintain compatibility with the MULTIBUS design frame, the coprocessor design frame provides a completely synchronous internal interface to the custom circuit. A two-phase non-overlapping clock is provided for the custom circuit to use. The clock is derived from the external clock used by the coprocessor interface. In some microprocessor systems, this clock may be different from the main processor's clock, such as the Motorola 68020. In others, like the Intel 8086, the coprocessor clock is the same as the system clock. However, this difference is not apparent to the custom circuit since the coprocessor design frame for a specific microprocessor family provides clock circuitry and synchronizing logic to hide the difference.

When access to the system bus or service by the main processor is required, the custom circuit must use a synchronous request/acknowledge transaction protocol to issue the request to the coprocessor design frame. The coprocessor design frame translates

this simple protocol into the correct sequence of signals to gain control of the bus or request service from the main processor.

The coprocessor design frame automatically latches the system address and data busses when required and provides the custom circuit with these signals on unidirectional busses. This configuration is consistent with the MULTIBUS design frame. The MULTIBUS design frame allows the custom circuit to access both memory and I/O space on the system bus, but the coprocessor interface for most microprocessors only supports memory accesses. Therefore, only memory addressing is supported within the coprocessor design frame. The width of data transfers in the coprocessor design frame is fixed at 16 bits as is the MULTIBUS design frame.

There are two areas in which the coprocessor design frame differs significantly from the MULTIBUS design frame. These areas are interrupts and master/slave compatibility. The MULTIBUS design frame provides circuitry to implement an interrupt status register and to poll and clear interrupt requests. Due to the nature of some coprocessor interfaces, in particular the Intel 8086/8088, interrupts from multiple coprocessors are not entirely supported and, in others, not at all. Therefore, the coprocessor design frame does not provide any support for an interrupt line. However, if an interrupt request line is available, the coprocessor design frame provides that line directly to the custom circuit through synchronizing logic.

The MULTIBUS design frame provides circuitry which allows the custom circuit to act as either a master or slave on the system bus, but with a coprocessor interface, the coprocessor monitors the instructions which the main processor is executing and takes action when a coprocessor instruction is executed. The main processor generally treats a coprocessor instruction as a no-operation, except for decoding the addressing and

doing a dummy read from memory if required. Thus, a command contained within the coprocessor instruction may be given to the custom circuit along with any data from the dummy read. The coprocessor design frame provides the transaction circuitry mentioned earlier which allows the custom circuit to request access to the system bus if more data is required. Therefore, the coprocessor design frame differs from the MULTIBUS design frame by providing lines to indicate that a coprocessor instruction has been encountered and what the instruction is. These lines do not have to be used by the user's circuit, but they add an extra dimension to the capabilities of the design frame.

2.4 Internal Functional Description

With the general internal protocol defined, the internal operation and interface signals presented to the custom circuit can be described for the coprocessor design frame. A consistent notation for signals is used throughout the remainder of this thesis. An active low signal is indicated by a line over the signal name. Thus, a signal named *READ* is active high and a signal named $\overline{WRITE}$ is active low.

Although there are I/O coprocessors available for various microprocessor families, the coprocessor design frame presented here is designed for memory addressing. To maintain compatibility with the MULTIBUS design frame, the address space available to the custom circuit is 20 bits wide with the capability to be expanded to the size of the microprocessor bus on which it resides. The coprocessor design frame can respond to any address within this address space. Since the decision was made to provide only 16-bit data transactions, the lowest order address bit is not presented to the custom circuit. When programs which access the coprocessor design frame are written, all references

to addresses used by the coprocessor design frame should be on 16-bit boundaries since most microprocessors provide byte addressing capability.

As with the MULTIBUS design frame, the internal signals of the coprocessor design frame can be grouped based on function. Please note that if a signal present in the coprocessor design frame is the same as one in the MULTIBUS design frame, the nomenclature from the MULTIBUS design frame is used[7]. These groups are:

- Clocks and Initialization Lines – *PHI1, PHI2, INIT*
- Memory Access Control Lines – *MRD, MWR, MACK*
- Extended Bus Control Lines – *GETB, HAVEB, ORQST, RELB*
- Interrupt Lines (if present) – *INTR*
- Coprocessor Access Control Lines – *CIN, CRD, CACK, COMP*
- Coprocessor Command Lines – *CMD0-CMD15*
- Data Lines – *DATI0-DATI15, DATO0-DATO15*
- Address Lines – *ADRI1-ADRI19, ADRO1-ADRO19*

As with the MULTIBUS design frame, any unused signals should be tied low unless stated otherwise.

2.4.1 Clocks and Initialization Lines

The clock lines, *PHI1* and *PHI2*, provide the user's custom circuit with a two-phase non-overlapping clock which is synchronized with an external system clock. All

interface signals are synchronous to this two-phase clock with inputs expected to be valid on *PHI1* and outputs changed on *PHI2* to maintain compatibility with the MULTIBUS design frame. The initialization line, *INIT*, is asserted for at least one clock cycle to provide proper system startup at power-up or when a system reset is issued and is the same as in the MULTIBUS design frame.

2.4.2 Memory Access Control Lines

The memory access control lines, *MRD*, *MWR*, and *MACK*, are the handshake lines provided for single-word data transfers. The transaction protocol for these signals and any other signals requiring a two-way handshake is the same as the transaction protocol for the MULTIBUS design frame shown in Figure 2.3 and described below. Memory Read, *MRD*, is asserted for a minimum of one clock cycle by the user's circuit when a single-word bus read is desired. Memory Write, *MWR*, is asserted for a minimum of one clock cycle by the user's circuit when a single-word bus write is desired. Memory Acknowledge, *MACK*, is asserted by the coprocessor design frame when the read or write initiated by the user's circuit has been completed.

2.4.3 Extended Bus Control Lines

The extended bus control lines are provided for compatibility with the MULTIBUS design frame. Some microprocessor families provide the capability of alternate bus masters while others do not. The coprocessor design frames which can gain control of the system bus will use these lines as in the MULTIBUS design frame. But, the coprocessor design frames which cannot gain control of the system bus will simulate the desired responses and use only single-word transfers to accomplish the task. Get Bus

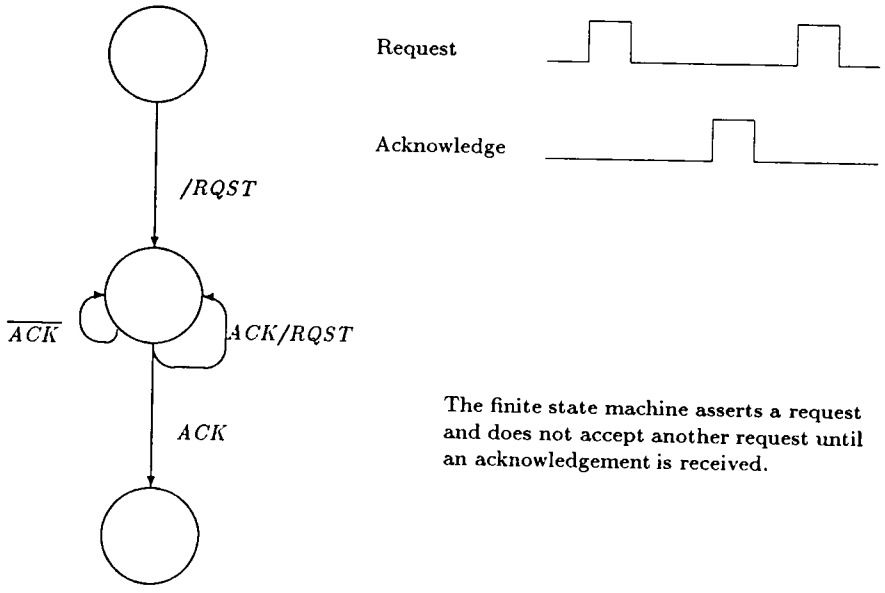


Figure 2.3: Transaction protocol (After Borriello et al. [7] page 11).

Control, *GETB,* is asserted for a minimum of one clock cycle by the user's circuit when a bus request is desired. Have Bus Control, *HAVEB,* is asserted when the coprocessor design frame has acquired the bus and is held for the duration of the control. This signal is asserted regardless of whether the transaction was initiated by *MRD, MWR,* or *GETB.* The coprocessor design frame asserts Other Bus Request, *ORQST,* when another device is requesting control of the bus from the user's circuit. This signal is also asserted regardless of whether the transaction currently occurring was initiated by *MRD, MWR,* or *GETB.* Finally, the user's circuit must assert Release Bus Control, *RELB,* when the extended bus transaction is completed. These signals are entirely compatible with those of the MULTIBUS design frame, and as such, *GETB* should be asserted either before or at the same time as the first *MRD* or *MWR* to gain extended control of the system bus.

2.4.4 Interrupt Line

The Interrupt Line, *INTR,* is provided to the user's circuit to initiate an interrupt on those microprocessor families which allow coprocessor interrupts. On these systems, the user's circuit asserts *INTR* for the duration of the interrupt request and the coprocessor design frame only synchronizes it to the system clock. The user's circuit must provide some means to service and clear the interrupt. One method would be to have a coprocessor instruction defined to service and clear the interrupt. When the coprocessor interface does not have an interrupt capability, the signal is not present.

2.4.5 Coprocessor Access Control Lines

The coprocessor access control lines, *CIN, CRD, CACK,* and *COMP,* are the handshake lines provided for the coprocessor interface. Coprocessor Instruction, *CIN,* is asserted for one clock cycle by the coprocessor design frame to indicate that a coprocessor instruction has been encountered by the main processor. Coprocessor Read, *CRD,* is asserted by the coprocessor design frame to indicate that a dummy memory read cycle has been initiated by the main processor as a result of the coprocessor instruction. The data from this dummy read is made available to the user's circuit for the instruction being executed. Coprocessor Acknowledge, *CACK,* is asserted for a minimum of one clock cycle by the user's circuit to indicate that the coprocessor instruction has been received. These signals, *CIN, CRD,* and *CACK,* operate on the same transaction protocol as *MRD, MWR,* and *MACK.* Coprocessor Complete, *COMP,* is asserted by the user's circuit for a minimum of one clock cycle to indicate that the current coprocessor instruction has completed execution.

2.4.6 Coprocessor Command Lines

The coprocessor command lines, *CMD0-CMD15,* provide the user's circuit with the coprocessor instruction that was encountered by the main processor. Due to coprocessor interface differences, some coprocessor design frames may not implement the full 16-bit command field as a useable range. In these cases, those command lines not implemented will be the higher-order bits and will be tied low by the design frame. The user's circuit is responsible for the decoding of the instruction.

2.4.7 Data Lines

The data lines in the coprocessor design frame are compatible with the lines in the MULTIBUS design frame. They are unidirectional and allow only word-oriented accesses. The data-in lines, *DATI0-DATI15,* provide the user's circuit with the data obtained either by a read request or a dummy read by the main processor (*MRD* or *CRD*). The data-out lines, *DATO0-DATO15,* provide the coprocessor design frame with the data which the user's circuit wants output during a write request (*MWR*).

2.4.8 Address Lines

The address lines in the coprocesssor design frame are also compatible with the lines in the MULTIBUS design frame. They are unidirectional and the low-order bit is lost since only word-addressing is supported. The address-in lines, *ADRI0-ADRI19,* provide the user's circuit with the address of the dummy read performed by the main processor in response to memory reference coprocessor instructions (*CRD*). The address-out lines, *ADRO0-ADRO15,* provide the design frame with the address to be used for the data transaction requested by the user's circuit (*MRD* or *MWR*).

2.5 Design Frame Operation

To effectively use a design frame, the designer must understand how to logically interface to the design frame. The coprocessor design frame provides a very simple interface for the designer to understand. Initialization of the custom circuit should be performed when *INIT* is asserted. The design frame derives this signal from the system reset or initialization signal.

Certain signals within the coprocessor design frame require a two-way handshaking for proper communication. The transaction protocol is the same as that implemented by the MULTIBUS design frame. The request signal is asserted for one clock cycle. When the transaction is completed, the acknowledge signal is asserted for one clock cycle. The coprocessor design frame translates this protocol into whatever is required for the external bus. The signals which use this transaction protocol are *CIN* and *CRD* with *CACK* and *MRD* and *MWR* with *MACK*.

Of particular interest to a designer is how data transactions are accomplished. The coprocessor design frame has address and data latches to capture and hold the information. As with the MULTIBUS design frame, all outgoing addresses and data must be valid during the clock cycle that the request or acknowledge is asserted. All incoming addresses and data are valid from when the request or acknowledge is asserted to the beginning of the next transaction.

To maintain greater compatibility with the MULTIBUS design frame, the coprocessor design frame provides signals (*GETB*, *HAVEB*, *ORQST*, and *RELB*) which allow the user's custom circuit to acquire the external bus for a set of uninterrupted data transactions. Some coprocessor interfaces, such as the Intel 80286 coprocessor interface, do not allow external bus control. For this type of interface, the signals are provided as if the capability did exist, but the transactions are actually a series of single-word transactions. This is easily accomplished since the signals which are used for the single-word transactions are used during a multi-word transaction to indicate when an read or write should be executed. Detailed timing diagrams are provided for all the above operations in Figures 2.4-2.8.

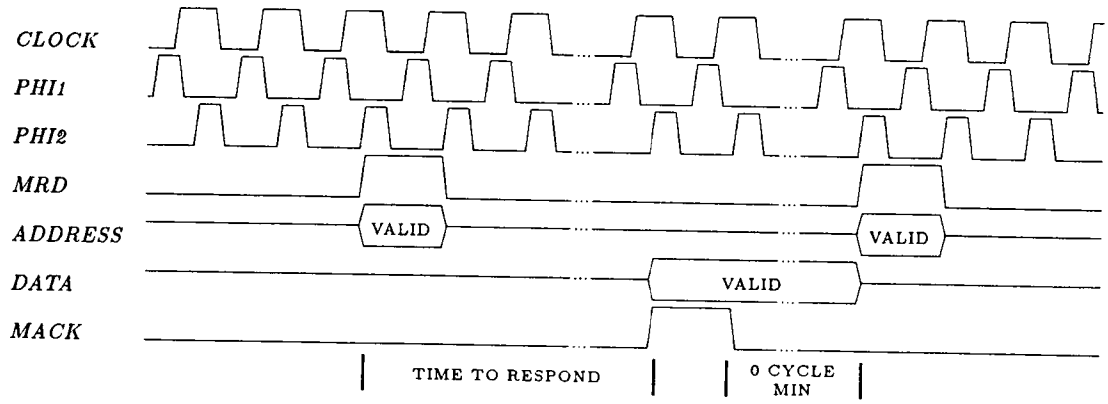


Figure 2.4: Memory read timing (After Borriello et al. [7] page 14).

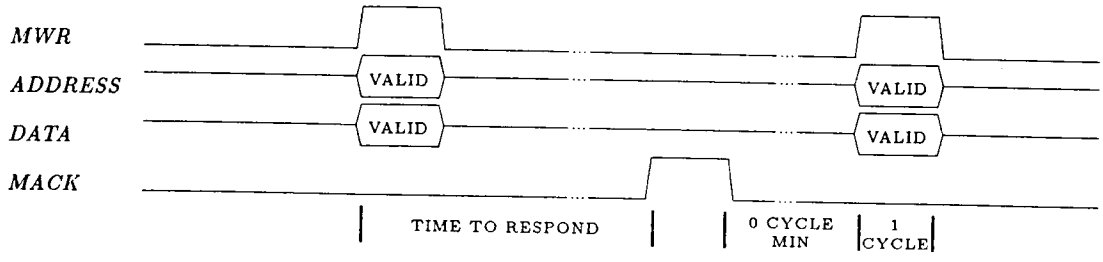


Figure 2.5: Memory write timing (After Borriello et al. [7] page 15).

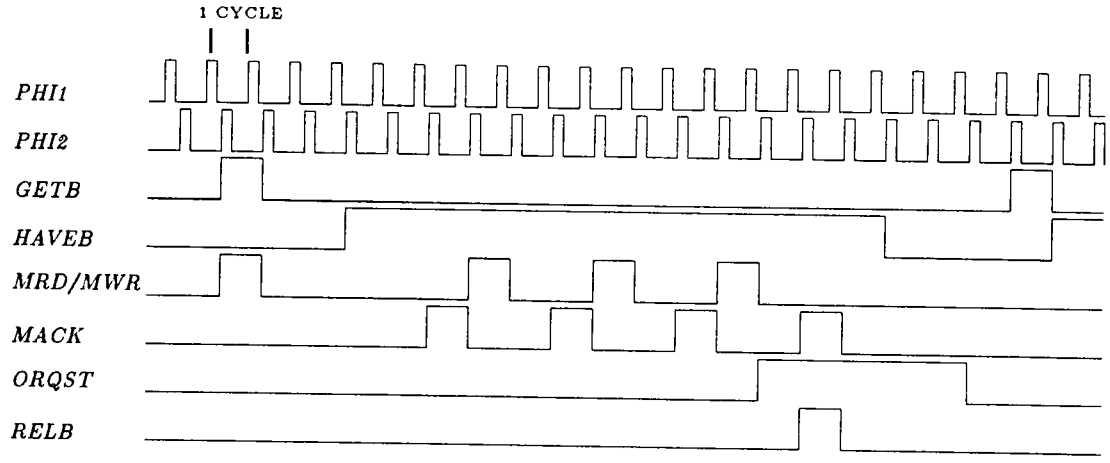


Figure 2.6: Extended bus use timing (After Borriello et al. [7] page 15).

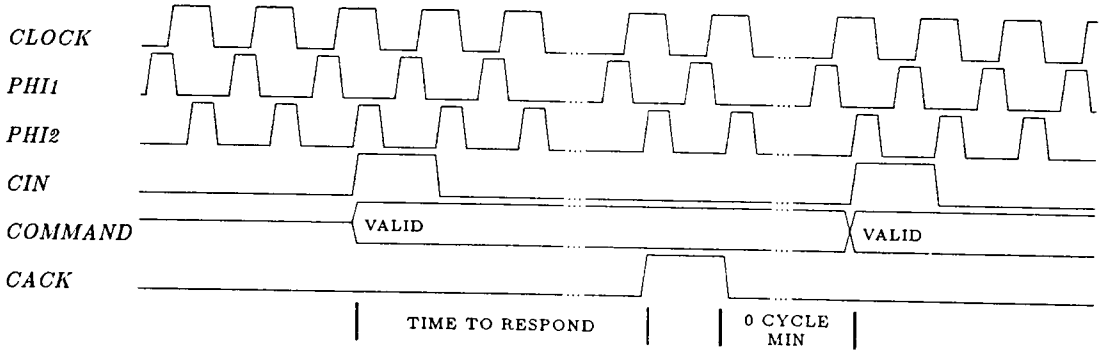


Figure 2.7: Command read timing.

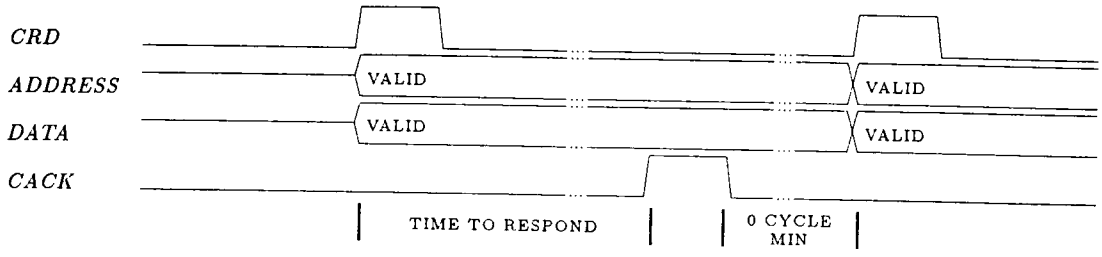


Figure 2.8: Coprocessor data read timing.

CHAPTER 3

COPROCESSOR CAPABILITIES

3.1 Coprocessor Interfaces

The microprocessors considered for a coprocessor design frame are the Motorola 68020, the Intel 8086/8088, and the Intel 80286. Each of these microprocessors has a coprocessor interface which is reasonably defined and a numeric coprocessor which is available. Microcomputer systems which use these microprocessors are also available and many provide a prewired socket for a numeric coprocessor. To design a coprocessor design frame, the coprocessor interface for a microprocessor family must be understood.

3.1.1 Motorola 68020 Coprocessor Interface

The 68020 is currently the most powerful microprocessor available from Motorola. It is a full 32-bit implementation of the M68000 family of microprocessors and is compatible with the earlier members of the M68000 family. The 68020 has an asynchronous bus structure using non-multiplexed address and data lines with a dynamic bus sizing capability. With the introduction of the 68020, Motorola defined the 68020 coprocessor interface for the M68000 family of microprocessors. The 68020 has microcode to implement the coprocessor interface, but other microprocessors, including non-M68000 family microprocessors, would need instruction sequences written that would emulate the coprocessor interface protocol. Of the three coprocessor interfaces considered, the 68020 coprocessor interface is the most well defined and provides considerable expansion

capabilities. The 68020 is currently used in several engineering workstations such as the Sun-3 workstation[1].

Most of the following information concerning the 68020 coprocessor interface was obtained from an excellent user's manual for the 68020 which contains a full description of the coprocessor interface and is available from Motorola[3]. The 68020 coprocessor interface has many capabilities. Physically, there are no special signals for a coprocessor to generate since standard M68000 asynchronous bus cycles are used to transfer information between the main processor and the coprocessor. Since communication is performed using an asynchronous bus, the coprocessor does not have to operate at the same clock frequency as the main processor. Thus, the speed of the coprocessor can be chosen to obtain optimum performance. Of particular interest to a coprocessor designer is the fact that a coprocessor must conform only to the coprocessor interface and an extensive knowledge of the microprocessor architecture is not required.

The 68020 coprocessor interface is designed to provide full support for non-concurrent operation between the main processor and coprocessor. Concurrent operation is possible, but the programming model should be based on sequential, non-concurrent instruction execution. The design of a coprocessor defines the instruction set for that coprocessor. A 68020 coprocessor instruction is identified by the use of the F-line operation words in the M68000 instruction set. The operation word is the first word of any M68000 family instruction. The F-line operation word has ones in bits 15 through 12 (Figure 3.1) and the remaining bits indicate the coprocessor instruction. Additionally, extensions words providing more information for the coprocessor may follow the F-line operation word. As shown in Figure 3.1, a coprocessor identification code (Cp-ID) is

[1]Sun-3 is a trademark of Sun Microsystems, Inc.

part of the F-line operation word. This allows multiple coprocessors to exist on the same microprocessor bus. User defined coprocessors should use the coprocessor identification codes 110-111 which do not interfere with current and future Motorola coprocessors.

A group of interface registers, shown in Figure 3.2, are defined for the 68020 coprocessor interface. The communication protocol needed to execute a coprocessor instruction is based on these registers. The 68020 uses standard M68000 asynchronous bus cycles to access the registers. The bus interface unit for the coprocessor must satisfy the 68020 address, data, and control timing to insure proper communication with the main processor. During a coprocessor instruction, the registers are accessed by the 68020 executing bus cycles to CPU space which is indicated by three function code lines being driven high. Thus, the coprocessor interface register set appears to the 68020 as any other peripheral interface register set does.

During a coprocessor access, the function code lines and address bus of the 68020 are used to select which coprocessor is being accessed. The register which is being accessed is also indicated on the address bus. Figure 3.3 shows the information present on the function code lines and address lines of the 68020 during a coprocessor access. Address lines *A19-A16* indicate the CPU space type which is 0010 for a coprocessor access, and address lines *A15-A13* indicate the coprocessor identification code. Figure 3.4 shows a block diagram of the signals involved in the 68020 coprocessor interface.

There are four categories of coprocessor instructions defined for the 68020 coprocessor interface. These are general, conditional, context save, and context restore. The categories are determined by the type of operation performed by the coprocessor instructions contained in them. The category also determines the coprocessor interface register which is accessed. The general and conditional categories are used to request

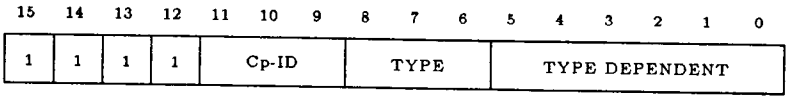


Figure 3.1: F-line coprocessor instruction operation word (After Motorola [3] page 8-3).

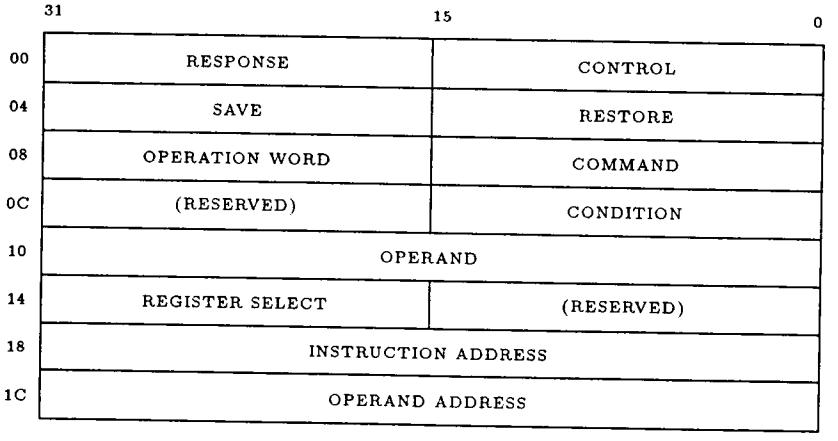


Figure 3.2: Coprocessor interface registers (After Motorola [3] page 8-6).

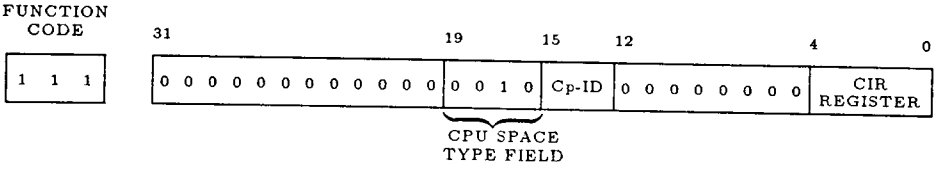
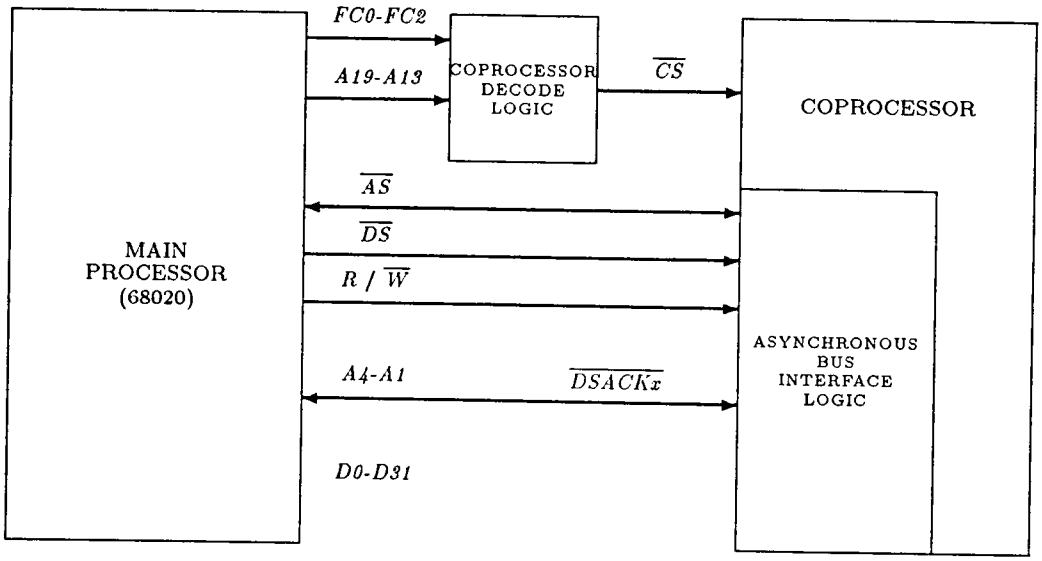


Figure 3.3: 68020 CPU space address encodings (After Motorola [3] page 8-5).



FC2-FC0 = 111 → CPU Space Cycle

A19-A16 = 0010 → Coprocessor Access in CPU Space

A15-A13 = xxx → Coprocessor Identification

A4-A1 = rrrr → Coprocessor Interface Register Selector

Address lines not specified above are "0" during coprocessor access.

Figure 3.4: 68020 coprocessor interface signal usage (After Motorola [3] page 8-7).

services and indicate status to the main processor using a set of predefined response primitive codes. The context save and restore categories are used to indicate status and enable restarting the coprocessor at a given point. All of the instruction categories are presented in great detail in the 68020 user's manual and have a predefined protocol which can be implemented by a finite-state machine. However, a coprocessor need only support those instruction types which it defines and does not have to define addressing modes since the main processor can perform all data transfers. The 68881 is a numeric coprocessor available from Motorola which serves as a very good coprocessor example using the 68020 coprocessor interface.

3.1.2 Intel 8086/8088 Coprocessor Interface

The Intel 8086/8088 is actually two microprocessors, the 8086 and the 8088. These microprocessors are identical except that the 8086 is designed for a 16-bit bus and the 8088 is designed for an 8-bit bus. Thus, the coprocessor interface is defined for both the 8086 and 8088. The 8086/8088 is one of the most widely used microprocessors currently available. In large part, this is due to the prevalence of the IBM-PC/XT and its many clones.

The 8086/8088 uses a time-multiplexed address and data bus and can operate in either a minimum or maximum mode. The mode identifies whether the processor is configured for a small, single-procesor system (minimum mode) or a medium or large system with coprocessors or multiple processors (maximum mode). The coprocessor interface is defined only for maximum mode systems and is not as well defined as the M68000 coprocessor interface.

Most of the following information concerning the Intel 8086/8088 and its coprocessor

interface was obtained from the user's hardware reference manual for the 8086/8088 and is available from Intel[2]. A coprocessor for the 8086/8088 must interface to the same bus as the processor which includes the address and data lines, status, clock, ready, reset, test, and request and grant signals and meet the timing requirements for these signals. The 8086/8088 coprocessor interface requires that the designer of a coprocessor understand the 8086/8088 and its bus structure.

With the 8086/8088 coprocessor interface, the coprocessor must track the execution of instructions by the 8086/8088 and determine when a coprocessor instruction has been encountered. The 8086/8088 has an instruction queue allowing instructions to be prefetched and causing the tracking of instruction execution to be more difficult. However, the 8086/8088 provides two queue status lines to indicate the action performed on the instruction queue and function lines to indicate when an instruction is being fetched. From these signals, the coprocessor can track the status of the 8086/8088 instruction execution completely. A significant difference between the 8086 and the 8088 is that the 8086 has a six byte queue and the 8088 has a four byte queue. A coprocessor determines which processor, an 8086 or 8088, is on the bus by monitoring a status line, $\overline{BHE}$ / *S7*, after a reset.

The 8086/8088 coprocessor interface implements a coprocessor instruction through the use of the ESC instruction to the 8086/8088. The main processor and coprocessor both decode and execute the ESC instruction. The main processor will treat the ESC as a no-operation but will distinguish between those that reference memory and those that do not. If the instruction references memory, the main processor will decode the addressing mode and perform a dummy read at the calculated address. Thus, the coprocessor is able to identify the instruction to be performed and receive any additional

information through the dummy read. Note that the address for the dummy read is available on the system bus for capture and storage by the coprocessor. An additional line on the main processor, *TEST*, is driven by the coprocessor to indicate when the coprocessor instruction has been completed. The WAIT instruction of the 8086/8088 tests this line and proceeds if the instruction has been completed. If *TEST* indicates that the coprocessor instruction has not completed, the 8086/8088 is placed in a wait state until the instruction is completed.

Access to the system bus for more data is provided to the coprocessor through a set of bus grant and request lines connected to the main processor and other local devices. The 8086/8088 coprocessor interface can gain complete control of the local system bus through these lines. Thus, the 8086/8088 coprocessor interface has the advantage of allowing a coprocessor to become the bus master and essentially taking over execution of instructions if desired. No coprocessor for the 8086/8088 currently does this, but it is possible. The Intel 8087 numeric coprocessor is a very good example of an implementation of the 8086/8088 coprocessor interface.

3.1.3 Intel 80286 Coprocessor Interface

The Intel 80286 is an advanced microprocessor with specialized capabilities for multiple user and multi-tasking systems. The 80286 is upward software compatible with the Intel 8086/8088, but the coprocessor interface (called a processor extension interface by Intel) is distinctly different. The 80286 is gaining use in more microcomputer systems, in particular the IBM-PC/AT[2] and clones.

Most of the following information concerning the Intel 80286 and its coprocessor

[2]IBM-PC/AT is a trademark of the International Business Machines Corporation (IBM).

interface was obtained from a components hardware reference manual which is available from Intel[4]. It is highly recommended that an up-to-date manual be obtained due to the discovery that earlier 80286 literature is in error, particularly in reference to the coprocessor interface.

Although the 80286 is upward software compatible with the 8086/8088, the 80286 differs significantly in hardware capabilities from the 8086/8088. The 80286 has separate address and data busses and does not provide a queue status output as does the 8086/8088. The 80286 coprocessor interface could be described as being somewhere between the 8086/8088 coprocessor interface and the M68000 coprocessor interface. The 80286 has microcode to implement the coprocessor interface like the Motorola 68020, but the 80286 coprocessor interface is not as versatile as the M68000 coprocessor interface. Functionally, the 80286 coprocessor interface is more like the 8086/8088 coprocessor interface. However, the definition of the 80286 coprocessor interface is not as well documented as the other two coprocessor interfaces. This is hopefully due to the relative infancy of the 80286.

The 80286 handles all the addressing, instruction fetching, and data transfers for a coprocessor. This information is passed to the coprocessor through what is called the processor extension data channel. Unlike the 8086/8088 coprocessor interface, the 80826 coprocessor interface does not require the coprocessor to determine when a coprocessor instruction has been encountered. The 80286 will transmit the coprocessor instruction to the coprocessor through the processor extension data channel when one is encountered. If the coprocessor instruction referenced memory, the 80286 will transmit this data as well. The coprocessor can request the 80286 to perform data transfers only after a coprocessor instruction has been encountered. The 80286 coprocessor interface imple-

ments a coprocessor instruction through the use of the 80286 ESC instruction which is the same as the 8086/8088 instruction.

There are specific signal lines available on the 80286 to implement the processor extension data channel, but there is no capability currently available for a coprocessor to gain control of the system bus like the 8086/ 8088 coprocessor interface. The current documentation on the 80286 does not provide an adequate description of the action of the processor extension signal lines during execution of a coprocessor instruction. However, the Intel 80287 numeric coprocessor is an example of an implementation of the 80286 coprocessor interface.

3.2 Coprocessor Selection

Design frames implementing the above coprocessor interfaces would give a VLSI designer the capability to test and evaluate a custom VLSI design very quickly with an existing microprocessor system. As noted earlier, conceptual designs for each coprocessor interface are to be presented. However, a decision was made to design completely only one at this time.

The key factors in the decision were based on certain design constraints presented earlier. The design constraints involved in the decision were universality, standardization, electrical specifications, and assembly. Of the coprocessor interfaces, the M68000 coprocessor interface is the most well defined and is extremely well documented. However, microcomputer systems which contain the Motorola 68020 are not as numerous as those of the Intel 8086/8088 (universality). The Intel 80286 suffers from the same problem. In particular, the fact that microcomputer systems containing the Intel 8086/8088

were readily available and had a prewired socket for a coprocessor (assembly) had the greatest influence on the decision. This is based on the fact that the main function of a design frame is the rapid prototyping, testing, and evaluation of a VLSI design, and if a system to test the design is not available, the design becomes a tie tack. However, there were other factors which supported the decision.

One requirement which was placed on the coprocessor design frame was that it must be pin compatible with the commercially available coprocessor so that it could be placed in the prewired socket. The Motorola 68881 numeric coprocessor is contained in a 68-pin grid-array which is available from MOSIS but only as a special request which may require a longer turnaround time. The Intel 8087 and 80287 numeric coprocessors are 40-pin DIPs which are available as a standard package from MOSIS. However, the standard 40-pin package available from MOSIS has pin-1 connected to the substrate and should be connected to the lowest voltage present. Pin-1 on the Intel 80287 is specified as a "not connected" line and would require a special wire bonding operation to correctly connect the signals. This could also require a longer turnaround time. The Intel 8087, which has pin-1 connected to ground, meets the requirements for a standard 40-pin pad frame and package available from MOSIS (standardization, electrical specifications, and assembly).

A desired feature of the coprocessor design frame is the capability for a custom design to gain control of the system bus which allows a much wider range of custom designs (universality). The Intel 8086/8088 coprocessor interface provides this capability and maintains pin compatibility with its commercial coprocessor, the Intel 8087. The M68000 coprocessor interface does allow the coprocessor to access the system bus, but the Motorola 68881 does not have that capability. Since all data transfers are required to

be performed by the main processor, the Intel 80286 coprocessor interface does not allow the coprocessor to gain access to the system bus. Thus, the Intel 8086/8088 coprocessor interface, which the Intel 8087 implements, provides the most desirable framework for the testing and evaluation of a coprocessor design frame.

CHAPTER 4

MOTOROLA 68020 COPROCESSOR DESIGN FRAME

4.1 General M68000 Coprocessor Design Frame Considerations

The Motorola 68020 coprocessor interface is very versatile, but a complete implementation of the interface is not required for a coprocessor to be used by the 68020. A coprocessor needs only to implement that portion of the 68020 coprocessor interface which is required for the coprocessor to function as intended. A very detailed description of the 68020 coprocessor interface cannot be presented here due to the large amount of information concerning it. However for anyone who desires more information, the user's manual for the 68020 contains nearly 60 pages of very detailed information about the 68020 coprocessor interface[3]. Note that other microprocessors could use the coprocessor, but code to emulate the interface protocol would be required.

A coprocessor design frame can be designed which implements a complete 68020 coprocessor interface. But, the circuit area required would be large and the custom circuit would need much more complex interface circuitry. A minimum coprocessor implementation can be designed using only three or five of the 68020 coprocessor interface registers (CIRs) and maintain the simple internal protocol to the custom circuit. For this minimum implementation, the custom VLSI design which is placed in the coprocessor design frame must fit one of two categories. These categories are that either the custom circuit always generates the addressing for required data or that the custom circuit does no addressing and instead has the 68020 generate the addressing. The first

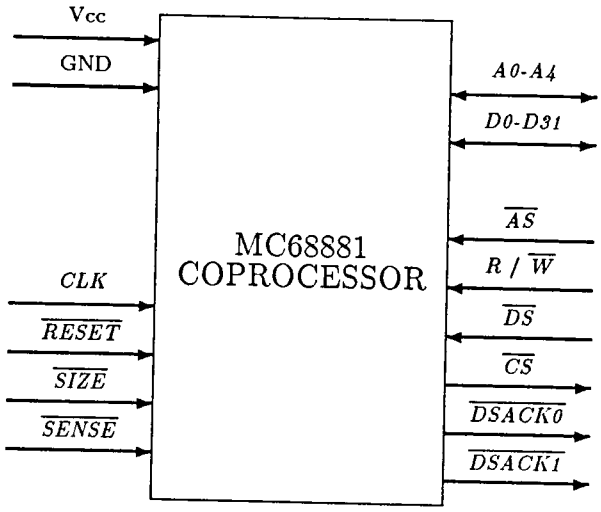


Figure 4.1: 68881 coprocessor signals.

type of circuit is similar to a microprocessor and allows the circuit to skip over sections of code and data as needed. The second type of circuit can be considered as a "true" coprocessor-type circuit and is the easiest to implement using the 68020 coprocessor interface. Both designs are presented here and adhere to the internal protocol presented earlier.

A requirement for both design frames is that the design frame be pin compatible with the Motorola 68881 numeric coprocessor. Custom VLSI circuits could then be used by a 68020 through a prewired 68881 numeric coprocessor socket. Figure 4.1 shows the 68881 and the signals required for interfacing. The 68881 does not have the capability to generate an interrupt to the 68020, therefore neither design frame supports interrupts generated by the custom circuit.

The main tasks of the design frame are to determine when a coprocessor instruction

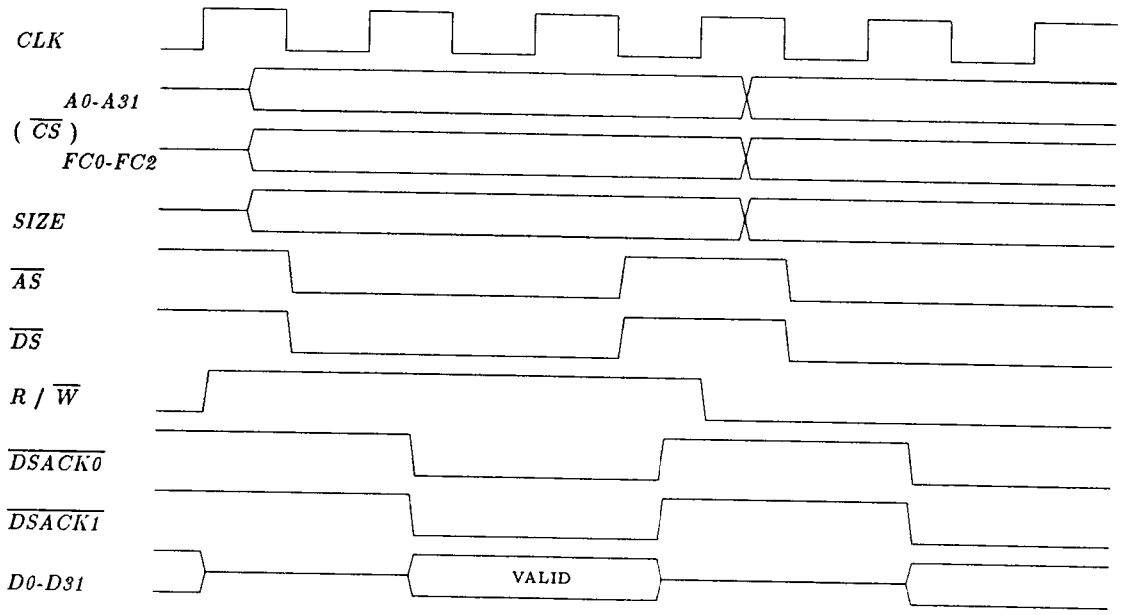


Figure 4.2: 68020 read cycle timing (After Motorola [3] Foldout 1).

for this coprocessor has been encountered and to interface communications between the custom circuit and the 68020. To perform these tasks, the design frame must interface to the 68020 asynchronous bus. The timing diagrams for bus cycles pertaining to the 68881 are shown in Figures 4.2 and 4.3. From the timing diagrams and information on bus operation in the user's manual for the 68020, the sequence for bus cycles can be described.

For a bus read, i.e. data transferred from the coprocessor to the main processor, the signals $\overline{AS}$, $\overline{DS}$, and $\overline{CS}$ must all be low to indicate a valid address is present on the external bus and that the main processor is waiting for data. When this occurs, the coprocessor must indicate the size of the data transfer with the signals $\overline{DSACK0}$ and $\overline{DSACK1}$. The activation of $\overline{DSACK0}$ and $\overline{DSACK1}$, which are normally in a

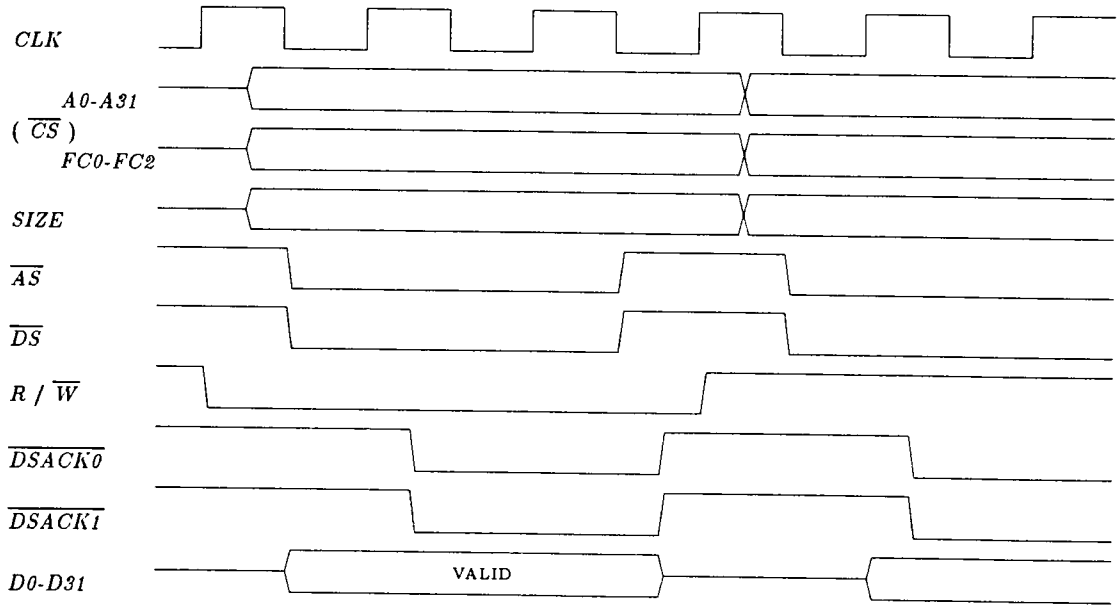


Figure 4.3: 68020 write cycle timing (After Motorola [3] Foldout 1).

Table 4.1: $\overline{DSACK}$ encodings.

Data Bus	A4	$\overline{DSACK1}$	$\overline{DSACK1}$	Comments
32-Bit	1	L	L	Valid Data on D0-D31
32-Bit	0	H	L	Valid Data on D16-D31
16-Bit	x	H	L	Valid Data on D16-D31
8-Bit	x	L	H	Valid Data on D24-D31
All	x	H	H	Insert Wait states in Current Bus Cycles

tri-state condition, also indicates to the main processor that the data is present on the external bus. Table 4.1 lists the encodings for $\overline{DSACK0}$ and $\overline{DSACK1}$. $\overline{DSACK0}$ and $\overline{DSACK1}$ are held until any one of the signals $\overline{AS}$, $\overline{DS}$, or $\overline{CS}$ is not low which indicates the main processor has latched the data and released the bus. At that time, $\overline{DSACK0}$ and $\overline{DSACK1}$ should be changed to indicate a wait cycle for one clock period and then tri-stated.

For a bus write, i.e. data transferred from the main processor to the coprocessor, the signals $\overline{AS}$, $\overline{DS}$, and $\overline{CS}$ must all be low to indicate a valid address and data is present on the external bus and that the main processor is waiting for the coprocessor to receive it. As with a bus read, the coprocessor must indicate the size of the data transfer with the signals $\overline{DSACK0}$ and $\overline{DSACK1}$. The activation of $\overline{DSACK0}$ and $\overline{DSACK1}$ also indicates to the main processor that the coprocessor has latched the data and the external bus can be released. $\overline{DSACK0}$ and $\overline{DSACK1}$ are held until any one of the signals $\overline{AS}$, $\overline{DS}$, or $\overline{CS}$ is not low which indicates the main processor

has released the bus. At that time, $\overline{DSACK0}$ and $\overline{DSACK1}$ should be changed to indicate a wait cycle for one clock period and then tri-stated.

Access to the 68020 asynchronous bus follows a logical sequence of steps, as does communication with the custom circuit and the 68020 coprocessor interface protocol. Thus, all of the functions of the design frame can be performed by a finite state machine. The block diagram for both 68020 coprocessor design frames is shown in Figure 4.4.

4.2 Design Frame with Addressing

The first 68020 coprocessor design frame, called M1, allows custom VLSI circuits to generate their own addressing from a starting address but does not control the bus directly. Instead, the design frame informs the 68020 of the address to be accessed and the 68020 performs the data transfer. This method may appear ackward, but it is very effective and has the added advantage that memory management protection, if active, is maintained by the 68020.

Two categories of 68020 coprocessor instructions are used to implement the above process – context restore and general instructions. The context restore coprocessor instruction is used to load the starting address of the code to be accessed by the custom circuit. The general coprocessor instruction category is used to pass the coprocessor command to the custom circuit, initiate execution of the command, and to perform any data transfers. The protocol required by the 68020 coprocessor interface for these two categories of coprocessor instructions is shown in Figures 4.5 and 4.6.

The context restore instruction is normally used to place a coprocessor into a specific state, but for M1 it is used to load the starting address of the code for the custom circuit.

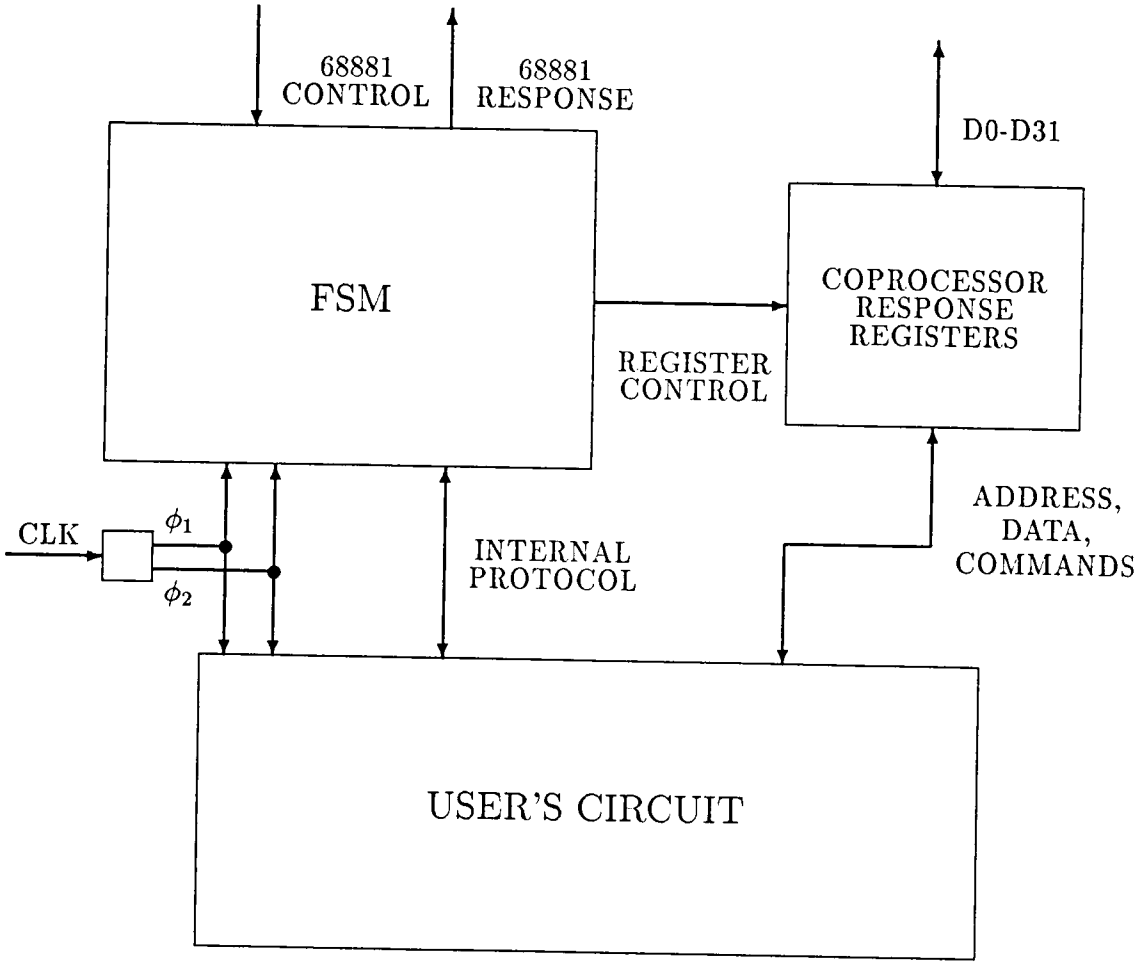


Figure 4.4: 68020 coprocessor design frame block diagram.

Main Processor		Coprocessor
M1 Recognize Coprocessor Instruction F-Line Operation Word		
M2 Write Coprocessor Command Word to Command CIR	→	C1 Decode Command Word and Initiate Command Execution
		C2 While (Main Processor Service is Required) do Steps (1) and (2) below
M3 Read Coprocessor Response Primitive Code from Response CIR	↔	(1) Request Service by Placing Appropriate Response Primitive Code in Response CIR
(1) Perform Service Requested by Response Primitive		(2) Receive Service from Main Processor
(2) If (Coprocessor Response Primitive Indicates "Come Again") go to M3 (see Note)		
		C3 Reflect "No Come Again" in Response CIR
		C4 Complete Command Execution
		C5 Reflect "Processing Finished" Status in Response CIR
M4 Proceed with Execution of Next Instruction		

NOTE: "Come Again" indicates that further service of the main processor is being requested by the coprocessor.

Figure 4.5: General Category instruction protocol (After Motorola [3] page 8-9).

Main Processor		Coprocessor
M1 Recognize Coprocessor Instruction F-Line Operation Word		
M2 Read Coprocessor Format Code from Effective Address Specified in Operation Word		
M3 Write Coprocessor Format Word to Restore CIR	→	C1 Terminate Current Operations and Evaluate Format Word
M4 Read Restore CIR	↔	C2 If (Invalid Format) Place Invalid Format Code in the Restore CIR
M5 If (Format = Invalid Format) Write $0001 Abort Code to Control CIR and Initiate Format Error Exception Processing		
M6 If (Format = Empty/Reset) go to M7 Else, Transfer Number of Bytes Specified by Format Word to Operand CIR (see Note)		C3 If (Valid Format) Receive Number of Bytes Indicated in Format Word Through Operand CIR
M7 Proceed with Execution of Next Instruction		

NOTE: The MC68020 uses the length field in the format word read during M2 to determine the number of bytes to read from memory and write to the operand CIR.

Figure 4.6: Context Restore instruction protocol (After Motorola [3] page 8-19).

Table 4.2: Context restore responses.

RESTORE CIR 15 7 0		
FORMAT CODE	LENGTH	MEANING
00	xx	Empty/Reset
01	xx	Not Ready, Come Again
02	xx	Invalid Format
03-0F	xx	Undefined Reserved
10-FF	Length	Valid Format, Coprocessor Defined

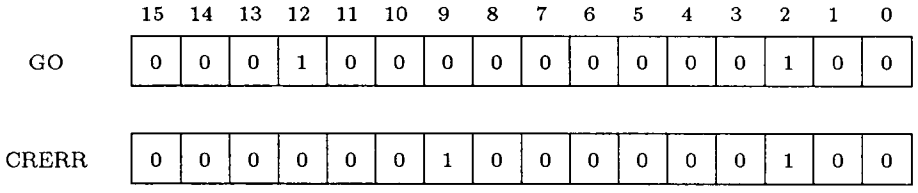


Figure 4.7: Design frame Context Restore responses.

When a context restore instruction is executed, several responses from a coprocessor are available (see Table 4.2). However for M1, only two responses are implemented – **GO** and **CRERR** (see Figure 4.7). **GO** indicates to the 68020 that everything is correct and to proceed with the data transfer. **CRERR** indicates that some type of error has occurred and to initiate error processing.

The general instruction is used to implement data processing instructions and other general purpose instructions for a coprocessor. For the general instruction, a set of

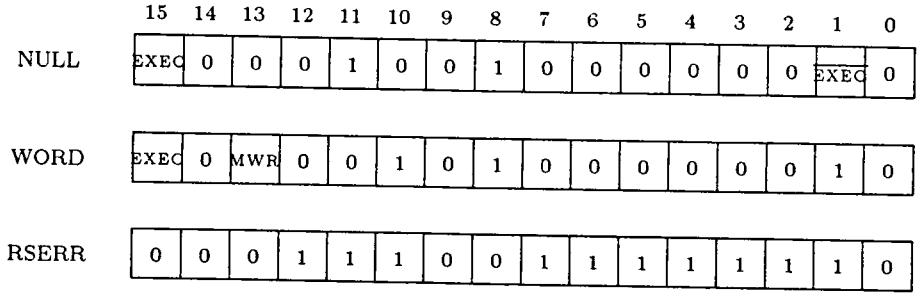


Figure 4.8: M1 primitive responses.

response primitives is defined. These response primitives are instructions that a coprocessor issues to the main processor to obtain various services. As with the context restore instruction, several responses from a coprocessor are available, but only three are implemented – **NULL**, **WORD**, and **RSERR** (see Figure 4.8). **NULL** indicates that the custom circuit has either completed processing or more processing is necessary. As long as **NULL** indicates that more processing is required, the 68020 will continue to check the response register and wait. **NULL** also indicates to the 68020 whether interrupts are to be serviced or not. For M1 **NULL** indicates that interrupts are to be serviced by the 68020.

WORD indicates to the 68020 that the custom circuit requires a data transfer and the direction of that transfer. The response primitive that **WORD** implements is the Take Address and Transfer Data primitive. Therefore, the 68020 will read the operand address register for the data address and then perform the necessary data transfer. The data address is provided by the custom circuit to the design frame through the normal use of *MRD* and *MWR*. **RSERR** indicates to the 68020 that some type of error has occurred and error processing should be initiated. In both **CRERR** and **RSERR**,

error processing is forced to indicate a format error.

The state diagram for the finite state machine in M1 is shown in Figure 4.9. Note that at any state, $\overline{RESET}$ will force a transition to the INIT state. Table 4.3 lists the states of M1, inputs required by each state, the active outputs of each state, and the possible next states for each state. Since neither design frame cannot actually gain control of the system bus for extended data transfers, the extended bus control signals to the custom circuit are simulated with hardware.

An additional aspect of the state diagram for M1 needs to be mentioned. The generation of the state diagram is based on correct observation of the coprocessor interface protocol by the 68020. By assuming that the 68020 will correctly implement the interface protocol, the state diagram is significantly simplified. In particular, several states can utilize the same next state and the WAIT state can be used to determine which portion of an instruction is being performed. If this assumption is not made, the number of the states would approximately double due to each instruction being a separate branch from the WAIT state and no two instructions using the same state below the WAIT state. The advantage to making the assumption is obvious since the area required by the finite state machine is greatly reduced. The disadvantage is that an error by the 68020 or other main processor in implementing the interface protocol could result in an improper state transition by the design frame. However, an error of this sort should be easily detected by the user and corrected with either a new 68020 or examination of the emulation code.

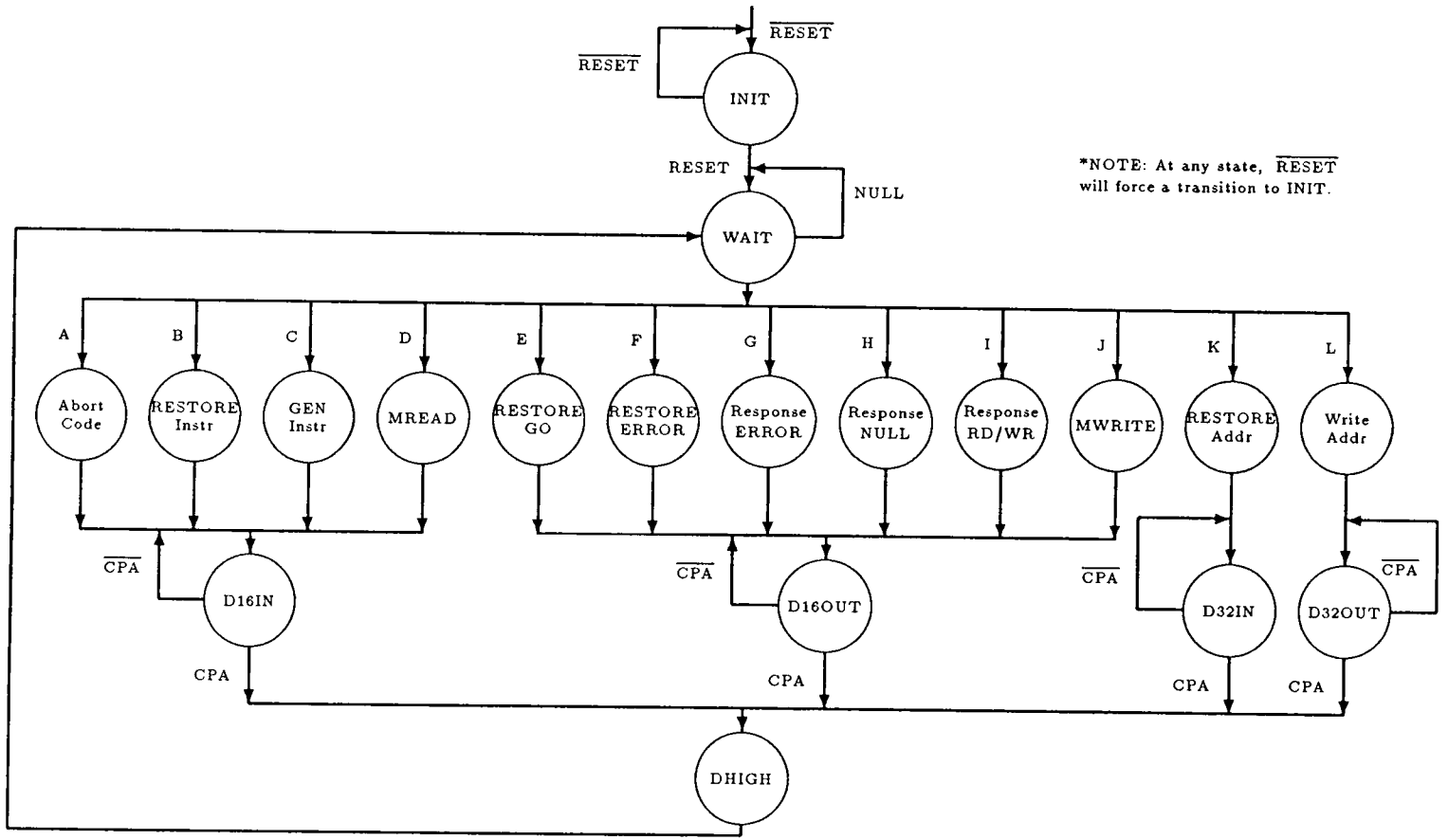


Figure 4.9: M1 state diagram.

Table 4.3: M1 state definitions.

STATE NUMBER	STATE NAME	INPUTS	OUTPUTS	NEXT STATES
1	INIT	$\overline{\text{RESET}}$	INIT, DZ	1, 4
2	WAIT – wait for a coprocessor instruction	NULL (CPA)	DZ, HZ	1, 2, 8-19
3	D16IN – 16-Bit data in	$\overline{\text{CPA}}$	D16, In	1, 7
4	D16OUT – 16-Bit data out	$\overline{\text{CPA}}$	D16, Out	1, 7
5	D32IN – 32-Bit data in	$\overline{\text{CPA}}$	D32, In	1, 7
6	D32OUT – 32-Bit data out	$\overline{\text{CPA}}$	D32, Out	1, 7
7	DHIGH – Tri-state I/O pads	None	DH, HZ	1, 2
8	RESTORE Instr – Context Restore coprocessor instruction execution. Accessing Restore CIR	(B) $\overline{\text{CPA}}$ ·(AD=06)· $\overline{\text{W}}$	D16, In	1, 3
9	RESTORE GO – Context Restore response indicating valid format	(E) $\overline{\text{CPA}}$ ·(AD=06)· R · nCACK	Lout, GO, Out, DZ	1, 4
10	RESTORE ERROR – Context Restore response indicating an error	(F) $\overline{\text{CPA}}$ ·(AD=06)· R · $\overline{\text{nCACK}}$	Lout, CRERR, Out, DZ	1, 4
11	RESTORE Addr – Starting address for custom circuit is loaded into Operand Address CIR	(K) $\overline{\text{CPA}}$ ·(AD=10)· $\overline{\text{W}}$ · nCACK · $\overline{\text{MEM}}$	Lin, AddrIn, CRD, In, DZ	1, 5
12	Abort Code – an error forces main processor to send an abort execution code to Control CIR	(A) $\overline{\text{CPA}}$ ·(AD=02)· $\overline{\text{W}}$	D16, DZ	1, 3
13	GEN Instr – General coprocessor instruction execution. Accessing Command CIR	(C) $\overline{\text{CPA}}$ ·(AD=0A)· $\overline{\text{W}}$	Lin, CIN, In, DZ	1, 3
14	Response ERROR – General instruction response indicating an error	(G) $\overline{\text{CPA}}$ ·(AD=00)· R · $\overline{\text{nCACK}}$	Lout, RSERR, Out,DZ	1, 4

*NOTE: CPA = $\overline{\text{AS}}$ + $\overline{\text{DS}}$ + $\overline{\text{CS}}$; AD is address lines A1-A4 ; MEM is MWR+MRD.

Table 4.3(cont.): M1 state definitions.

STATE NUMBER	STATE NAME	INPUTS	OUTPUTS	NEXT STATES
15	Response NULL – General instruction response indicating whether the custom circuit has completed execution or not	(H) $\overline{CPA} \cdot (AD=00) \cdot R \cdot nCACK \cdot \overline{MEM}$	Lout, NULL, Out, DZ	1, 4
16	Response RD/WR – General instruction response indicating a data transfer requested by the custom circuit	(I) $\overline{CPA} \cdot (AD=00) \cdot R \cdot \overline{nCACK} \cdot MEM$	Lout, WORD, Out, DZ	1, 4
17	Write Addr – data address for requested data transfer is read by main processor from the Operand Address CIR	(L) $\overline{CPA} \cdot (AD=1C) \cdot R \cdot \overline{nCACK} \cdot MEM$	Lout, AddrOut, Out, DZ	1, 6
18	MREAD – data for requested read data transfer is written by main processor to the Operand CIR	(D) $\overline{CPA} \cdot (AD=10) \cdot \overline{W} \cdot \overline{nCACK} \cdot MEM$	Lin, DataIn, MACK, In, DZ	1, 3
19	MWRITE – data for requested write data transfer is read by main processor from the Operand CIR	(J) $\overline{CPA} \cdot (AD=10) \cdot R \cdot \overline{nCACK} \cdot MEM$	Lout, DataOut, MACK, Out,DZ	1,4

*NOTE: CPA = $\overline{AS} + \overline{DS} + \overline{CS}$; AD is address lines A1-A4 ; MEM is MWR+MRD.

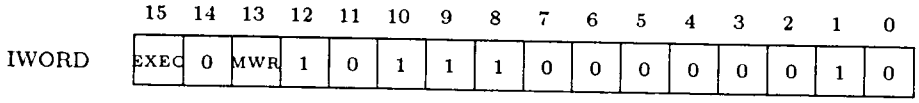


Figure 4.10: M2 IWORD primitive response.

4.3 Design Frame without Addressing

The second 68020 coprocessor design frame, called M2, does not allow custom VLSI circuits to generate their own addressing and assumes that the user has programmed the coprocessor instruction with an addressing mode which provides an auto-increment or -decrement feature. Due to the use of this type of addressing mode, the data for the design frame must be contiguous in memory and all the addressing responsibilites are left to the 68020. By permitting the 68020 to control addressing completely, M2 has faster data transfer rates than M1 since for each data transfer M1 must send an address to the 68020 and M2 does not.

Only one category of 68020 coprocessor instructions is used to implement the above process – general instructions. The general coprocessor instruction category is used to pass the coprocessor command to the custom circuit, initiate execution of the command, and to perform any data transfers. The protocol used by the general instruction category is the same as that used for M1 shown in Figure 4.5.

The general instruction used by M2 uses the same response primitives as M1 except for **WORD**. Instead of **WORD**, M2 will respond with **IWORD** (see Figure 4.10) which implements the response primitive Evaluate Effective Address and Transfer Data. Upon reading this primitive, the 68020 will evaluate the data address and perform the

data transfer based on the length passed in the primitive. **IWORD** always indicates a transfer size of two bytes for M2 since the design frame only transfers 16-bit words.

The state diagram for the finite state machine in M2 is shown in Figure 4.11. Note that at any state, $\overline{RESET}$ will force a transition to the INIT state. Table 4.4 lists the states of M2, inputs required by each state, the active outputs of each state, and the possible next states for each state. Note that the states of M2 are a subset of the states of M1 with the major difference being the elimination of any state dealing with addressing. As with M1, the state diagram of M2 has been significantly reduced by assuming that the 68020 or other main processor will correctly implement the interface protocol. Therefore, the statements concerning this assumption and any advantages and disadvantages apply to M2 as well as M1.

4.4 Software Considerations

A major consideration for any design frame is its actual use in an existing system. For either M1 or M2, this implies use within a 68020 system and the software to access the design frame. The first assumption that must be made is that the design frame is installed in the 68881 socket and has a coprocessor identification of 001. The software required to access M1 is presented first.

Two instructions are used to access M1 - cpRESTORE and cpGEN. The format of these instructions is shown in Figures 4.12 and 4.13 where Cp-ID is 001. To implement

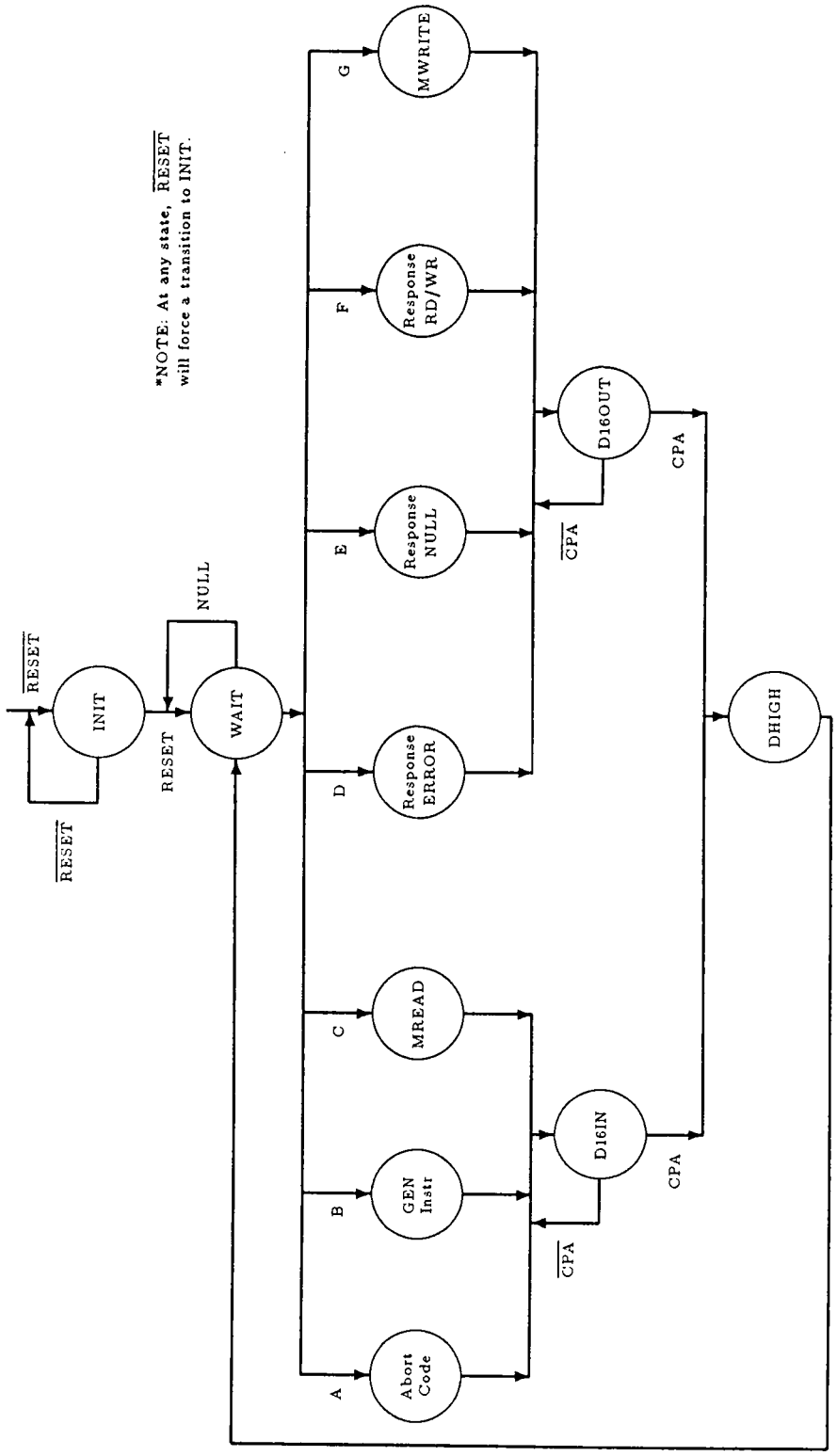


Figure 4.11: M2 state diagram.

Table 4.4: M2 state definitions.

STATE NUMBER	STATE NAME	INPUTS	OUTPUTS	NEXT STATES
1	INIT	$\overline{\text{RESET}}$	INIT, DZ	1, 4
2	WAIT – wait for a coprocessor instruction	NULL (CPA)	DZ, HZ	1, 2, 6-12
3	D16IN – 16-Bit data in	$\overline{\text{CPA}}$	D16, In	1, 5
4	D16OUT – 16-Bit data out	$\overline{\text{CPA}}$	D16, Out	1, 5
5	DHIGH – Tri-state I/O pads	None	DH, HZ	1, 2
6	Abort Code – an error forces main processor to send an abort execution code to Control CIR	(A) $\overline{\text{CPA}} \cdot$(AD=02)$\cdot \overline{\text{W}}$	D16, DZ	1, 3
7	GEN Instr – General coprocessor instruction execution. Accessing Command CIR	(B) $\overline{\text{CPA}} \cdot$(AD=0A)$\cdot \overline{\text{W}}$	Lin, CIN, In, DZ	1, 3
8	Response ERROR – General instruction response indicating an error	(D) $\overline{\text{CPA}} \cdot$(AD=00)$\cdot$ R $\cdot \overline{\text{nCACK}}$	Lout, RSERR, Out,DZ	1, 4
9	Response NULL – General instruction response indicating whether the custom circuit has completed execution or not	(E) $\overline{\text{CPA}} \cdot$(AD=00)$\cdot$ R $\cdot$ nCACK $\cdot \overline{\text{MEM}}$	Lout, NULL, Out, DZ	1, 4
10	Response RD/WR – General instruction response indicating a data transfer requested by the custom circuit	(F) $\overline{\text{CPA}} \cdot$(AD=00)$\cdot$ R $\cdot \overline{\text{nCACK}} \cdot$ MEM	Lout, IWORD, Out, DZ	1, 4
11	MREAD – data for requested read data transfer is written by main processor to the Operand CIR	(C) $\overline{\text{CPA}} \cdot$(AD=10)$\cdot \overline{\text{W}} \cdot \overline{\text{nCACK}} \cdot$ MEM	Lin, DataIn, MACK, In, DZ	1, 3
12	MWRITE – data for requested write data transfer is read by main processor from the Operand CIR	(G) $\overline{\text{CPA}} \cdot$(AD=10)$\cdot$ R $\cdot \overline{\text{nCACK}} \cdot$ MEM	Lout, DataOut, MACK, Out,DZ	1,4

*NOTE: CPA = $\overline{\text{AS}} + \overline{\text{DS}} + \overline{\text{CS}}$; AD is address lines A1-A4 ; MEM is MWR+MRD.

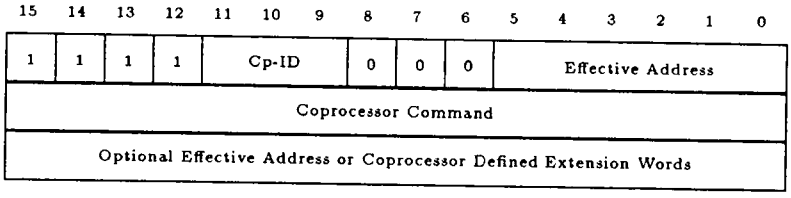


Figure 4.12: Coprocessor General instruction format (After Motorola [3] page 8-8).

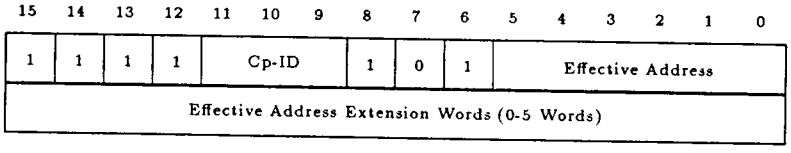


Figure 4.13: Coprocessor Context Restore instruction format (After Motorola [3] page 8-18).

a complete cycle to M1, the following sequence is performed:

```
cpRESTORE  StAddr
cpGEN      #COM
```

The cpRESTORE instruction transfers the starting address of the custom circuit's code to the design frame. StAddr represents the location where the starting address is stored. The cpGEN instruction transfers the command represented by COM to the design frame and then waits for completion while performing any data transfers for the custom circuit. Note that a command to the custom circuit does not necessarily imply any data transfers. Also, data transfers should be requested by the custom circuit

may be followed by several cpGEN instructions with different commands, but a cpGEN instruction should not occur without a cpRESTORE instruction somewhere before it.

Only one instruction is used to access M2 - cpGEN. The format of this instruction is the same as shown in Figure 4.12 and Cp-ID is 001. To implement a complete one cycle to M2, the following sequence is performed:

```
cpGEN     #COM
          or
cpGEN     #COM,(An)+
          or
cpGEN     #COM,-(An)
```

The first cpGEN instruction shown here transfers the command represented by COM to the design frame and then waits for completion. This instruction would be used if no data transfers are required by the custom circuit to execute the transferred command.

The second cpGEN instruction transfers the command represented by COM to the design frame and then waits for completion while performing any data transfers that are requested by the custom circuit. If any data transfers are requested, the 68020 will use the value contained in the address register, An, as the address of the data and then increment the value in the register by the data transfer size. In the case of M2, the data transfer size is always two bytes since the design frame is set to only transfer 16-bit words. Also, data transfers should be requested by the custom circuit only during the execution of a cpGEN instruction. The third cpGEN instruction performs exactly like the second instruction above except that the value in the address register is decremented

prior to the data transfer. In contrast to the M1 instructions, the instructions for M2 may occur in any sequence as long as the custom circuit has been designed to implement them.

CHAPTER 5

INTEL 80286 COPROCESSOR DESIGN FRAME

5.1 Hardware Design

The Intel 80286 coprocessor interface is not as well defined as the M68000 or Intel 8086/8088 coprocessor interfaces, but an overview of a design frame implementing the Intel 80286 coprocessor interface is presented. A major requirement for the design frame is that it be pin compatible with the Intel 80287 numeric coprocessor so that custom VLSI circuits could then be used by a 80286 through a prewired 80287 numeric coprocessor socket. Figure 5.1 shows the 80287 and the signals required for interfacing.

The Intel 80286 coprocessor interface does not allow the coprocessor to gain control of the system bus. Instead, the 80286 handles all data transfers for the coprocessor through the use of a processor extension data channel. The 80286 upon encountering an ESC instruction which indicates a coprocessor instruction transfers the instruction and any necessary data to the coprocessor via this processor extension data channel. Any request by the coprocessor for additional data or service must be made during the execution of a coprocessor instruction and is transferred via the processor extension data channel.

The 80286 has microcode which accesses the processor extension data channel through I/O port addresses 00F8(H), 00FA(H), and 00FC(H) which are part of the reserved port addresses by Intel. All I/O bus operations for a coprocessor are done through these port addresses. Only ESC instructions which reference memory enable the 80286 to accept

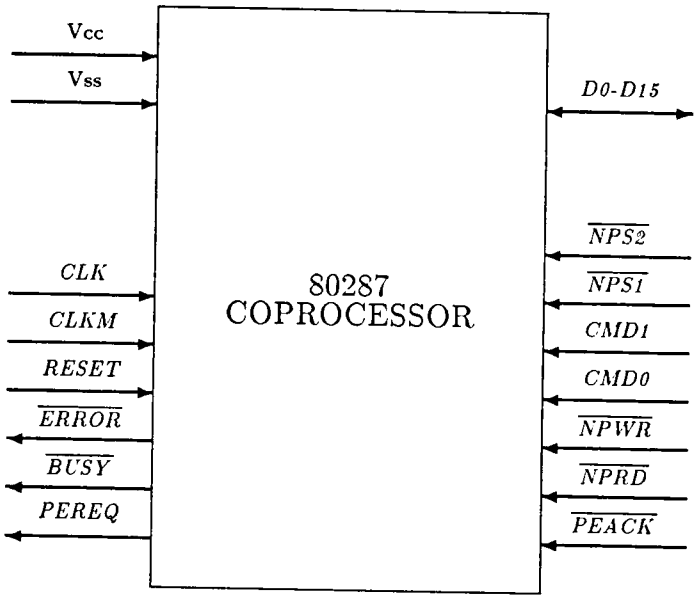


Figure 5.1: 80287 coprocessor signals.

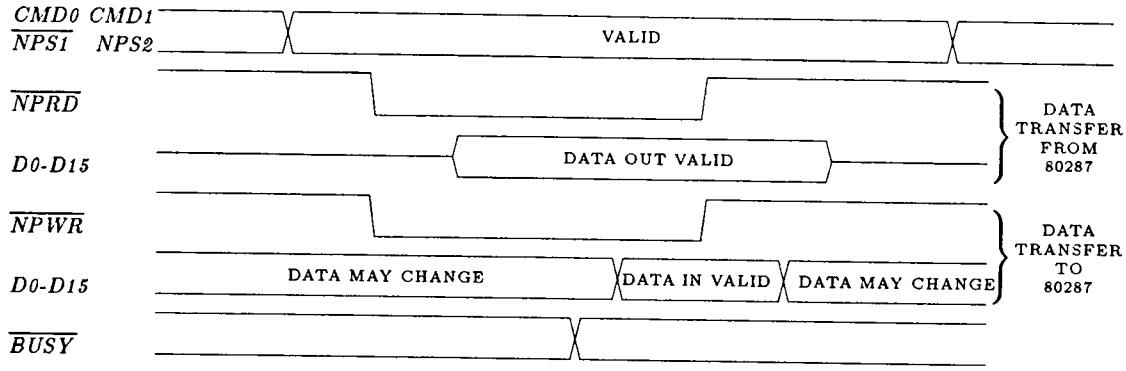


Figure 5.2: Data transfer timing initiated by 80286 (After Intel [4] page 4-75).

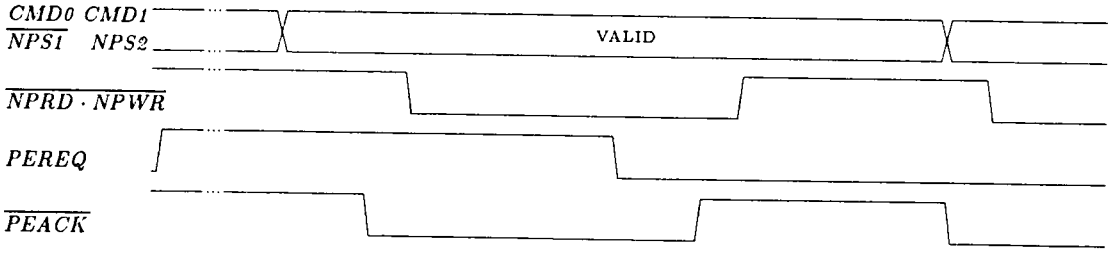


Figure 5.3: Data channel timing initiated by 80287 (After Intel [4] page 4-75).

requests by the coprocessor for data transfers. The 80286 determines from the ESC instruction the starting address for the data and whether it is a read or write. For each data transfer, two or three bus operations are performed, one word transfer with I/O port address 00FA(H) and one or two bus operations with memory. Thus, the 80286 always transfers 16-bit words to the coprocessor which is exactly what is desired for the design frame. The 80286 automatically performs the extra bus operations required when a word is aligned on an odd byte address. The timing diagrams for the processor extension data channel are shown in Figures 5.2 and 5.3.

The documentation available from Intel is not very specific about the sequence of events that occcurs during the execution of an ESC instruction. However, a general description can be assembled from various parts of the documentation[4]. When an ESC instruction is encountered by the 80286, the 80286 transfers the instruction opcode, the instruction address, and the data address, if the ESC instruction references memory, to the coprocessor through the processor extension channel. The order of these transfers and the sequence of control signals which indicate the transfers is not present in the current Intel documentation and would be required to do a complete design of an Intel 80286 coprocessor design frame. However, the documentation does indicate which signals are involved in any access to a coprocessor – *CMD0*, *CMD1*, $\overline{NPS1}$, and $\overline{NPS2}$.

If any data transfers occur, the signals $\overline{NPRD}$ and $\overline{NPWR}$ are used by the 80286 to indicate to the coprocessor the direction of the data transfer. The coprocessor uses the signal *PEREQ* to indicate that further data transfers are required and the 80286 will respond with $\overline{PEACK}$. The coprocessor asserts the signal $\overline{BUSY}$ to indicate that an instruction is being executed and releases it when processing is completed. Interrupts are not available to the coprocessor but an error can be indicated by the coprocessor by asserting the signal $\overline{ERROR}$. Since the design frame cannot gain control of the system bus, the extended bus signals from the custom circuit must be simulated with hardware.

Although complete knowledge of the signal sequences is not known, it is obvious that all of the functions of the design frame can be performed by a finite state machine. A detailed block diagram for the Intel 80286 coprocessor design frame, called I1, is shown in Figure 5.4 and Figure 5.5 shows the state diagram for the finite state machine. Note

that at any state, *RESET* will force a transition to the INIT state. Table 5.1 lists the states of the design frame, inputs required by each state, the active outputs of each state, and the possible next states for each state. Any inputs for which documentation is not fully available are indicated in this table.

5.2 Software Considerations

Use of I1 implies an 80286 system and software to access the design frame. The assumption is made that the design frame is installed in an 80287 numeric coprocessor socket. The ESC instruction is used by the 80286 to identify coprocessor instructions and, therefore, indicates the instructions used by the design frame as well.

The ESC instruction has the form shown in Figure 5.6 where TTT LLL are the opcode to the processor extension and mod and r/m form the normal 80286 addressing mode for the instruction. According to the documentation, the 80286 determines whether the instruction is a read or write, but how the 80286 determines this is not given. However by inspecting the instructions for the 80287 numeric coprocessor, a pattern to the processor extension opcode is apparent. The middle bit of LLL is always zero for a data transfer to the coprocessor from memory and always one for a data transfer from the coprocessor to memory. As a reasonable guess, the ESC instructions used to provide memory transfers for the design frame should use this same convention.

After initiating an ESC instruction, the 80286 continues program execution while a coprocessor executes the ESC instruction. This allows concurrent operation of the 80286 and a coprocessor. Synchronization is maintained by the 80286 by testing the $\overline{BUSY}$ pin before executing an ESC instruction to insure that the coprocessor has

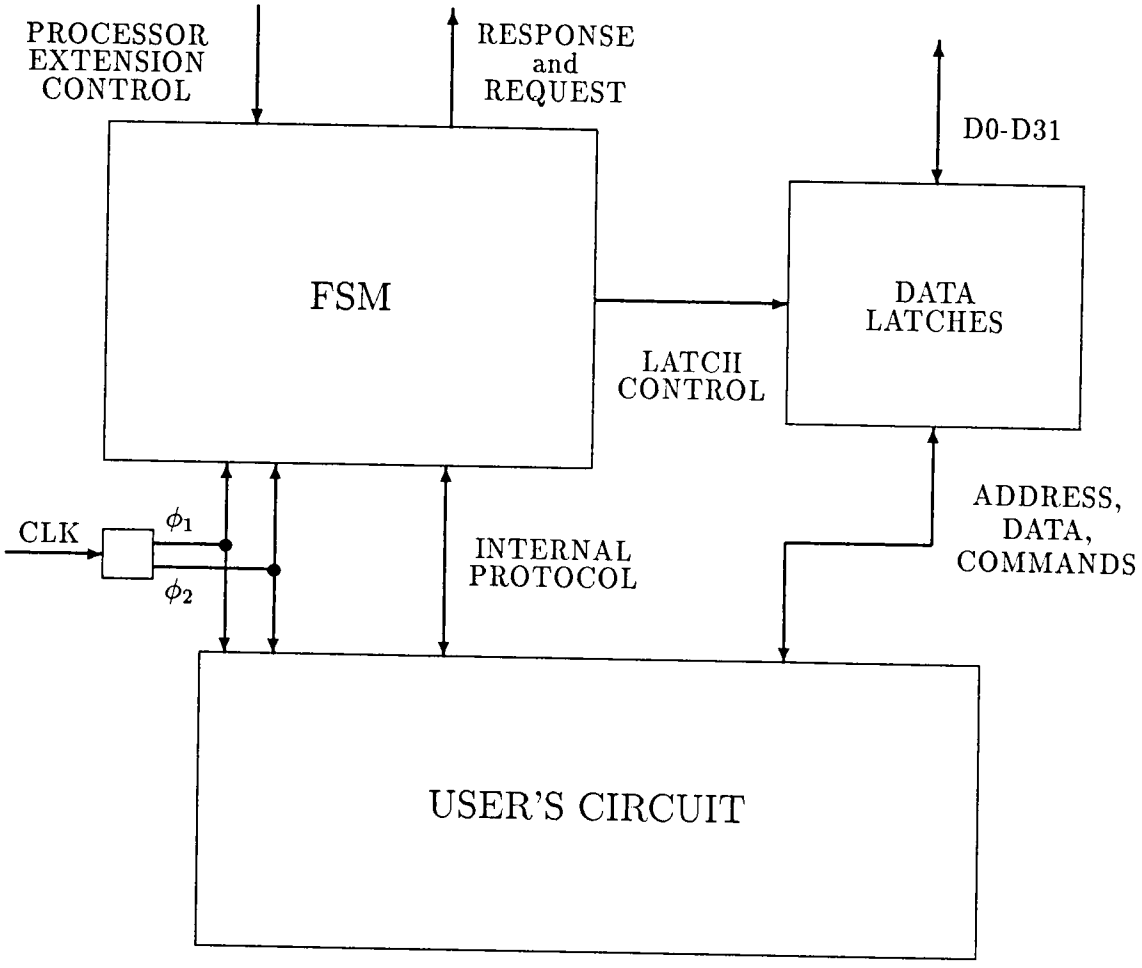


Figure 5.4: I1 block diagram.

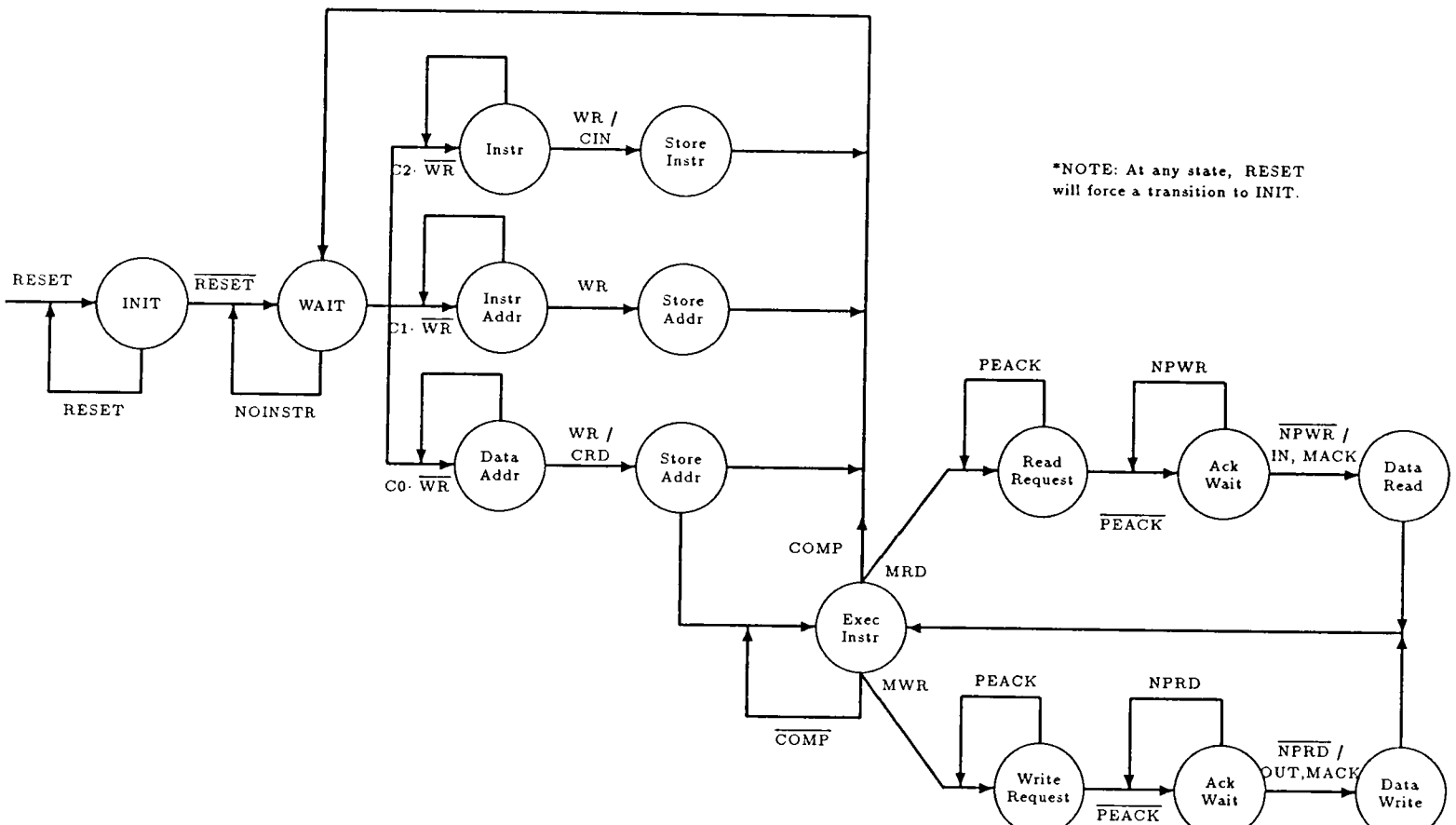


Figure 5.5: I1 state diagram.

Table 5.1: I1 state definitions.

STATE NUMBER	STATE NAME	INPUTS	OUTPUTS	NEXT STATES
1	INIT	$\overline{\text{RESET}}$	INIT	1, 2
2	WAIT – wait for a coprocessor instruction	$\overline{\text{NPS1}}$, NPS2, CMD0, CMD1, $\overline{\text{WR}}$	–	1, 2, 3-5
3	Instr – coprocessor instruction encountered	$\overline{\text{NPS1}}$, NPS2, CMD0, CMD1, $\overline{\text{WR}}$	–	1, 3, 6
4	Instr Addr – address of encountered coprocessor instruction being transferred	$\overline{\text{NPS1}}$, NPS2, CMD0, CMD1, $\overline{\text{WR}}$	–	1, 4, 7
5	Data Addr – address of coprocessor data (if present) being transferred	$\overline{\text{NPS1}}$, NPS2, CMD0, CMD1, $\overline{\text{WR}}$	–	1, 5, 8
6	Store Instr – get coprocessor instruction for custom circuit	WR	CIN	1, 2, 9
7	Store Addr – get coprocessor instruction address, but ignore it	WR	–	1, 2, 9
8	Store Addr – get coprocessor data address (if present) for custom circuit	WR	CRD	1, 2, 9
9	Exec Instr – execute coprocessor instruction	COMP	BUSY	1, 9, 10, 13
10	Read Request – read data requested by custom circuit	MRD, PEACK	PEREQ	1, 10, 11
11	Ack Wait – wait for data transfer acknowledge from main processor	$\overline{\text{PEACK}}$, NPWR	–	1, 11, 12
12	Data Read – read data requested by custom circuit and transferred by main processor	$\overline{\text{NPWR}}$	MACK, In	1, 9
13	Write Request – write data requested by custom circuit	MWR, PEACK	PEREQ	1, 13, 14
14	Ack Wait – wait for data transfer acknowledge from main processor	$\overline{\text{PEACK}}$, NPRD	–	1, 14, 15
15	Data Write – write data output by custom circuit and being transferred by main processor	$\overline{\text{NPRD}}$	MACK, Out	1, 9

*NOTE: The action of the signals $\overline{\text{NPS1}}$, NPS2, CMD0, and CMD1 are not completely documented in the Intel literature.

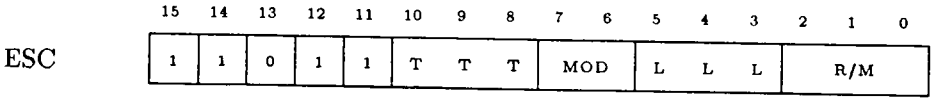


Figure 5.6: 80286 ESC instruction format (After Intel [4] page 4-52).

completed the previous ESC instruction. However, a WAIT instruction can be added to the code after the ESC instruction to essentially halt the 80286 until the ESC instruction has completed. This provides a means of forcing synchronization and of giving the coprocessor the full attention of the 80286.

In writing code for the 80286 which accesses the design frame, the user must know what the custom circuit requires and in what sequence for a given command to the custom circuit. The reason for this is that ESC instructions providing memory reads and writes must be executed when the custom circuit requires them due to the 80286 doing all the addressing. Thus, a typical code sequence for custom circuit which receives a command to perform an operation on some data and then output some resulting data is:

```
ESC     InitCom
ESC     Oper1,InAddr
ESC     Oper2,OutAddr
WAIT
```

The first ESC instruction transfers the command InitCom to the custom circuit which causes the custom cirucit to initialize itself. The next ESC instruction informs the

custom circuit to do the command Oper1 on the data starting at the address given by InAddr. Oper1 is defined to be a memory read operation so the custom circuit can perform only read operations during this instruction. The last ESC instruction informs the custom circuit to do Oper2 and output any data starting at the address given by OutAddr. Oper2 is defined to be a memory write so the custom circuit can perform only write operations during this instruction. The WAIT instruction is used to halt the 80286 until the custom circuit has finished processing and transfering all data.

The above sequence assumes a large data transfer in and out of the custom circuit, but similar sequences can be written which allow small data transfers. By doing small data transfers, memory reads and writes could be intermixed with greater frequency. Again, the sequence of ESC instructions must parallel the requirements for the custom circuit. The definition of the commands to the custom circuit which is TTT LLL in the ESC instruction is determined by the designer of the custom circuit. However as presented earlier, the middle bit must reflect the direction of the data transfer for a memory access instruction.

I1 is very much like the M68000 coprocessor design frame M2 since both are unable to do their own addressing. However by devising a set of commands to the custom circuit which transfers addresses between the custom circuit and the 80286 and the appropriate 80286 instructions, the custom circuit would be able to do addressing. In essence, a set of ESC instructions and other appropriate 80286 instructions could be written to create a similar structure as performed in the M68000 coprocessor design frame M1. The actual instructions are not presented here, but some of the ESC instructions can be described.

An ESC instruction could be defined which would contain a command to the custom

circuit which transfers an address as data from memory or a register in the 80286 to the custom circuit. The inverse of this would also be required so that the starting address for a data transfer could be transferred from the custom circuit to the 80286. Since the 80286 determines the direction of the transfer, a "transfer direction word" would need to be transferred between the custom circuit and the 80286 and, thus, an ESC instruction to do this. The actual data transfers could be performed in two subroutines, one for reading and the other for writing, where the ESC instruction used the address transferred earlier. A routine which looped until completion of a specific command to the custom circuit could be written to provide uninterrupted service. This is just the beginning of the possible software alternatives available to provide a broader capability to the custom circuit.

CHAPTER 6

INTEL 8086/8088 COPROCESSOR DESIGN FRAME

6.1 Hardware Design

The Intel 8086/8088 coprocessor interface presents a greater challenge to the designer than the other two coprocessor interfaces presented in this thesis. The M68000 and Intel 80286 coprocessor interfaces are defined such that the main processor informs the coprocessor that a coprocessor instruction has been encountered and what the instruction is as well as maintaining full bus control and providing a separate address and data bus. The designer using the 8086/8088 coprocessor interface must provide for tracking of the processor instruction stream, determining when a coprocessor instruction has been encountered, accessing the bus at the proper time for address and data, and controlling the system bus for the custom circuit. These problems are compounded due to the system bus being time-multiplexed between address and data.

Although the designer is presented with a greater challenge, the 8086/8088 coprocessor interface does provide the best framework for the testing and evaluation of a coprocessor design frame based on the advantages described in Chapter 3. A major requirement for the 8086/8088 design frame, called I2, is that it be pin compatible with the 8087 numeric coprocessor so that custom VLSI circuits can be used by a 8086/8088 through a prewired 8087 socket. Figure 6.1 shows the 8087 and the signals required for

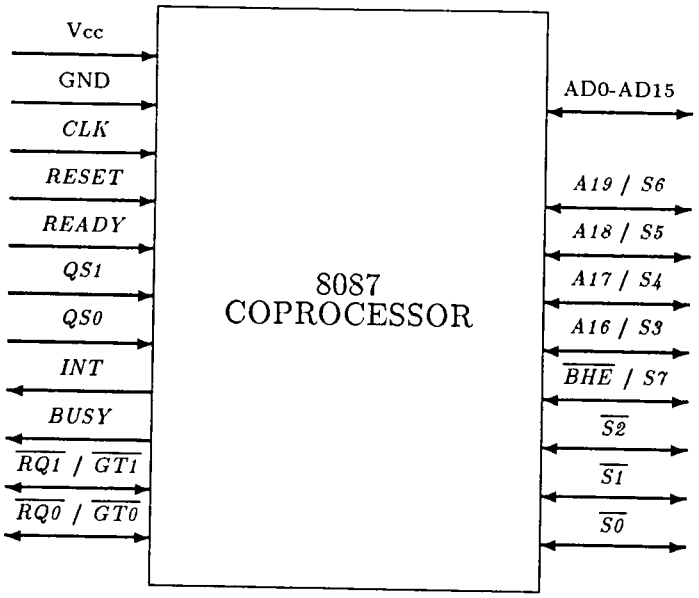


Figure 6.1: 8087 coprocessor signals.

interfacing. The 8087 can generate an interrupt so the signal *INTR* from the custom circuit is synchronized with the system clock and output on the interrupt line *INT*.

A brief overview of the 8086/8088 coprocessor interface was presented in Chapter 3, but a much more detailed description of the coprocessor interface and the 8086/8088 is required for the design and specification of a coprocessor design frame. The 8086/8088 does not have a mechanism like the 68020 or 80826 which informs the coprocessor that a coprocessor instruction has been encountered and what that instruction is. Instead, the coprocessor must track the instructions that the 8086/8088 is executing and determine for itself when a coprocessor instruction has been encountered. On the 8086/8088 the ESC instruction identifies a coprocessor operation. Thus, the operation of the 8086/8088 must be understood to design a coprocessor design frame.

The 8086/8088 is internally divided into two separate units, the bus interface unit

Table 6.1: 8086/8088 queue status lines (After Intel [4] page 3-220).

QS1	*QS0*	ACTION
0 (LOW)	0	No Operation
0	1	First Byte of Op Code from Queue
1 (HIGH)	0	Empty the Queue
1	1	Subsequent Byte from Queue

and the execution unit, which function in parallel. The bus interface unit handles all functions related to bus interface which includes prefetching instructions and operands, queuing instructions, and address relocation. The execution unit handles all functions related to data processing based on the prefetched instructions and operands.

Instructions are prefetched and stored in a first-in-first-out (FIFO) queue by the bus interface unit and then extracted by the execution unit when needed. The bus interface uses the three status lines $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ to identify the current bus operation and the execution unit uses the two queue status lines *QS0* and *QS1* to identify the queue operation. An instruction is placed in the queue by the bus interface unit during a bus cycle when $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ are 100, respectively. Table 6.1 lists the encoding of the queue status lines by the execution unit. Devices other than the 8086/8088 or the coprocessor can be the bus master and generate the signals $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$, but the 8086/8088 is the only device on the bus which drives *S6* low. The timing diagrams for bus cycles are shown in Figure 6.2. From the timing diagrams and information on bus operation from the microsystems components manual available from Intel[4], the sequence for bus cycles can be described.

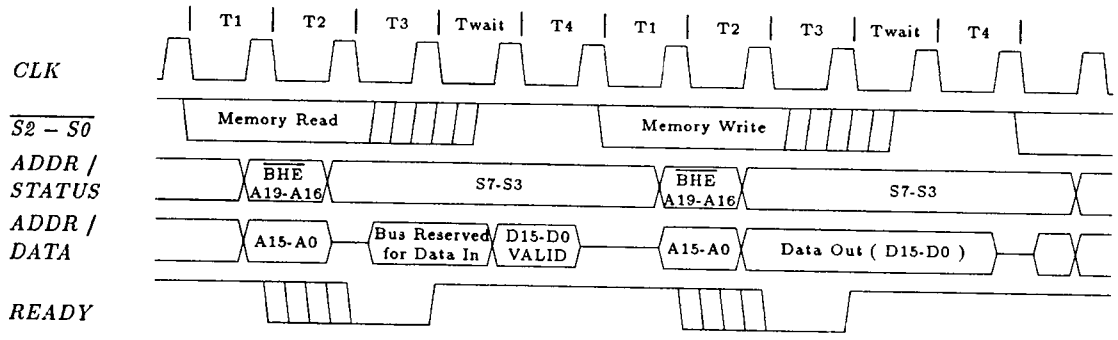


Figure 6.2: 8086/8088 basic system timing (After Intel [4] page 3-34).

The 8086/8088 executes a bus cycle to transfer data or fetch instructions. The only difference between an instruction fetch and a memory read is that the 8086/8088's bus interface places a different status on the bus. The minimum bus cycle consists of four processor clock cycles called T states. During the first T state (T1), the processor asserts an address and the status on the bus. For the second T state (T2), the processor removes the address from the bus and either tri-states its data outputs in preparation for a read cycle or asserts data for a write cycle. Also during T2, the upper four multiplexed bus lines switch from address (*A19-A16*) to bus cycle status (*S6, S5, S4, S3*). The status information is primarily diagnostic, except for *S6* which is driven low by the 8086/8088 and is used in systems to identify a processor bus cycle.

The status information asserted during T2 is continued during T3 and the processor will either continue to assert write data or sample read data on the data bus lines. If the selected memory or I/O device cannot transfer the data, the device must drive *READY* low which forces the processor to insert additional clock cycles (Wait states Twait) after T3. *READY* must be low by the start of T3. Bus activity during Twait

Table 6.2: $\overline{BHE}$ word transfer line (After Intel [4] page 3-2).

$\overline{BHE}$	$A0$	ACTION
0 (LOW)	0	Whole Word
0	1	Upper Byte from/to Odd Address
1 (HIGH)	0	Lower Byte from/to Even Address
1	1	None

is the same as T3. When the device has completed the transfer, it asserts *READY* and the processor will latch the data. If no wait states are inserted, the data is latched during T3. The bus cycle is terminated in T4 and the command lines are disabled. The 8086/8088 only executes a bus cycle when instructions or operands must be transferred to or from memory or I/O devices. Idle cycles (TI) are executed by the bus interface unit when not executing a bus cycle.

If all 16-bit words are even addressed, the 8086 transfers each word over the full 16-bit bus. However if a 16-bit word is on an odd address, the 8086 must perform two bus cycles and transfer the upper and lower bytes separately. The 8086 has an additional status line $\overline{BHE}$ / $S7$ to indicate that a word transfer is being performed at an odd address. Table 6.2 lists the activity of $\overline{BHE}$ and the lowest address bit $A0$ for the 8086. The 8088 does not have this situation since all data transfers are 8-bit and the equivalent status line on the 8088 is always high for a maximum mode system. $\overline{BHE}$ / $S7$ is also used to indicate which processor is on the bus by being low for the 8086 and high for the 8088 immediately after a reset.

Access and control of the system bus is achieved through the use of the $\overline{RQ}$ / $\overline{GT}$ lines. If the coprocessor desires control of the system bus or another device has requested control via the $\overline{RQ1}$ / $\overline{GT1}$ line, the $\overline{RQ0}$ / $\overline{GT0}$ line is driven low and then released. The processor grants control of the bus by then driving $\overline{RQ0}$ / $\overline{GT0}$ low. When the coprocessor is finished with the bus, $\overline{RQ0}$ / $\overline{GT0}$ is again driven low and released. The same type of protocol is followed when another device requests access to the bus via $\overline{RQ1}$ / $\overline{GT1}$. If the coprocessor has control of the bus, $\overline{RQ1}$ / $\overline{GT1}$ is driven low, released, and the system bus is released to the other device. The other device will drive $\overline{RQ1}$ / $\overline{GT1}$ low when access is finished and the coprocessor can again use the bus. If the coprocessor does not have control of the bus, a request on $\overline{RQ0}$ / $\overline{GT0}$ is issued and, when granted, passed to the other device. When the other device is finished, the release signal is passed on to the processor with $\overline{RQ0}$ / $\overline{GT0}$.

As with the other design frames, all functions of I2 can be performed by a finite state machine (FSM). However, the parallelism, multiplexed bus, and instruction tracking of the 8086/8088 creates a large number of states for the FSM. By splitting the tasks among multiple FSMs, the number of states required for each FSM becomes manageable. The block diagram for I2 is shown in Figure 6.3.

There are four FSMs in the design frame with three of the four FSMs performing a single task. By monitoring the status lines $\overline{S0}$, $\overline{S1}$, $\overline{S0}$, the CODE FETCH FSM identifies an instruction fetch cycle and latches the data while informing the QUEUE CONTROL FSM that an instruction is available. The QUEUE CONTROL FSM manages the instruction queue based on an instruction being added to the queue by the CODE FETCH FSM and the action specified by the queue status lines $QS0$ and $QS1$. The $\overline{RQ}$ / $\overline{GT}$ FSM handles the system bus request and grant protocol. The final

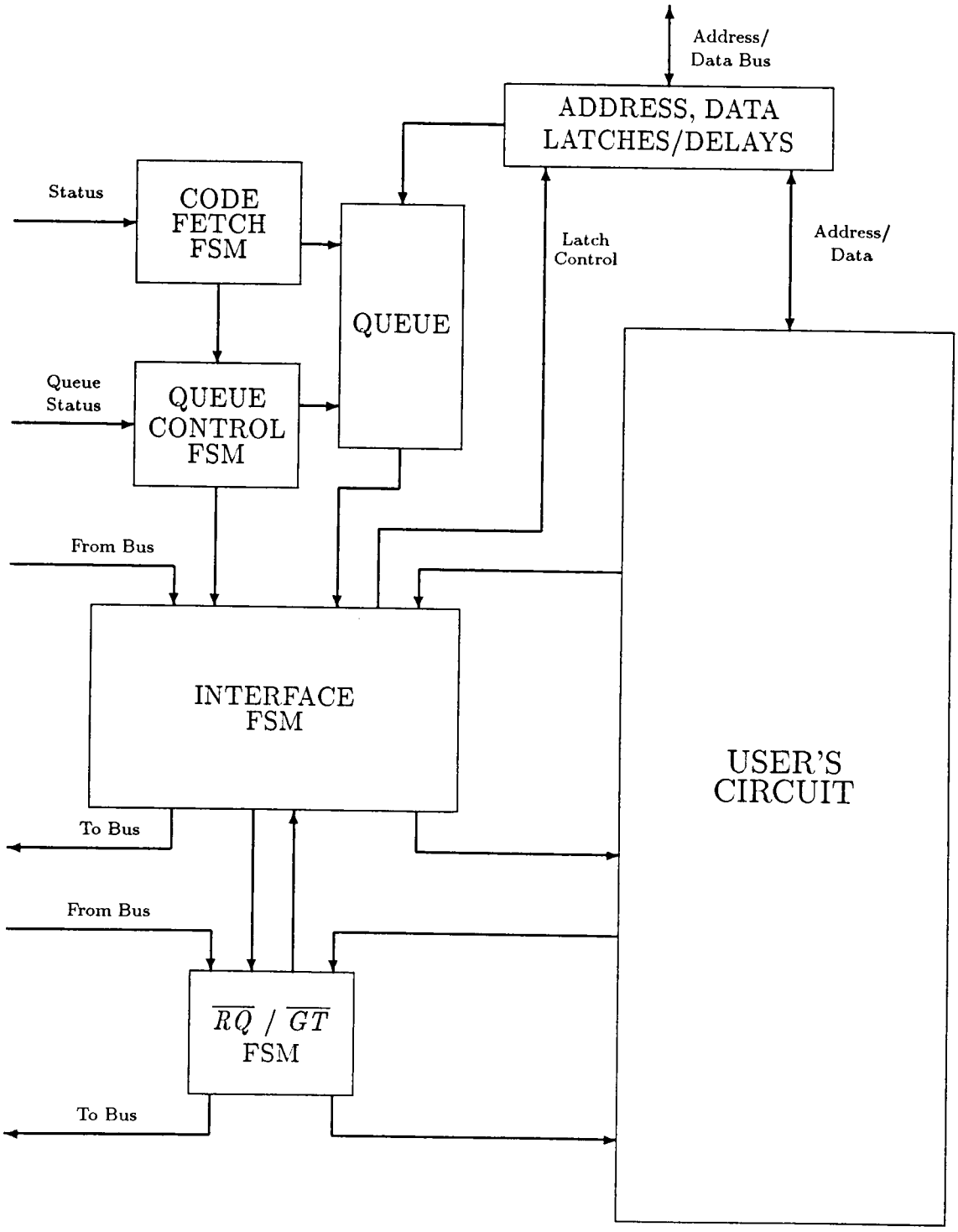


Figure 6.3: I2 block diagram.

FSM performs the remaining functions of the design frame which includes handling all communication with the custom circuit, determining if an instruction is an ESC instruction, capturing the address and data from a memory reference ESC instruction, informing the $\overline{RQ}$ / $\overline{GT}$ FSM to get control of the system bus, and handling control of the system bus during a memory read or write for the custom circuit. The state diagrams of all four FSMs are shown in Figures (6.4-6.7). Note that at any state, *RESET* forces a transition to the initial state for any of the FSMs.

To reduce the size of the FSMs and enable I2 to be used by both the 8086 and the 8088, several hardware additions and software assumptions are required. Since the status lines indicate a memory access during the same cycle that the address is available, a dynamic register is used to hold the address until the following FSM cycle so that the FSM can use the address or status. One use of this feature is to capture the address from a memory reference ESC instruction for use by the custom circuit. A six byte addressable register is used as the instruction queue. All control inputs from the custom circuit are latched with S-R flip-flops allowing multiple cycle use of the signals. The data latches handling I/O have two control signals which allow a full 16-bit word or only the upper byte of the 16-bit word to be latched. This allows I2 to be used either by the 8086 or the 8088. When a data transfer is performed by or for I2, the full 16-bit word is initially latched regardless of which processor is present. If the processor is a 8088, a second transfer which contains the upper byte is performed and the upper byte is latched. This assumes that the data is present only on even addresses and all data is 16-bit words which is a requirement presented in Chapter 2. Another hardware addition is the use of "bus-hold" circuitry (Figures 6.8 and 6.9) on the address and status lines.

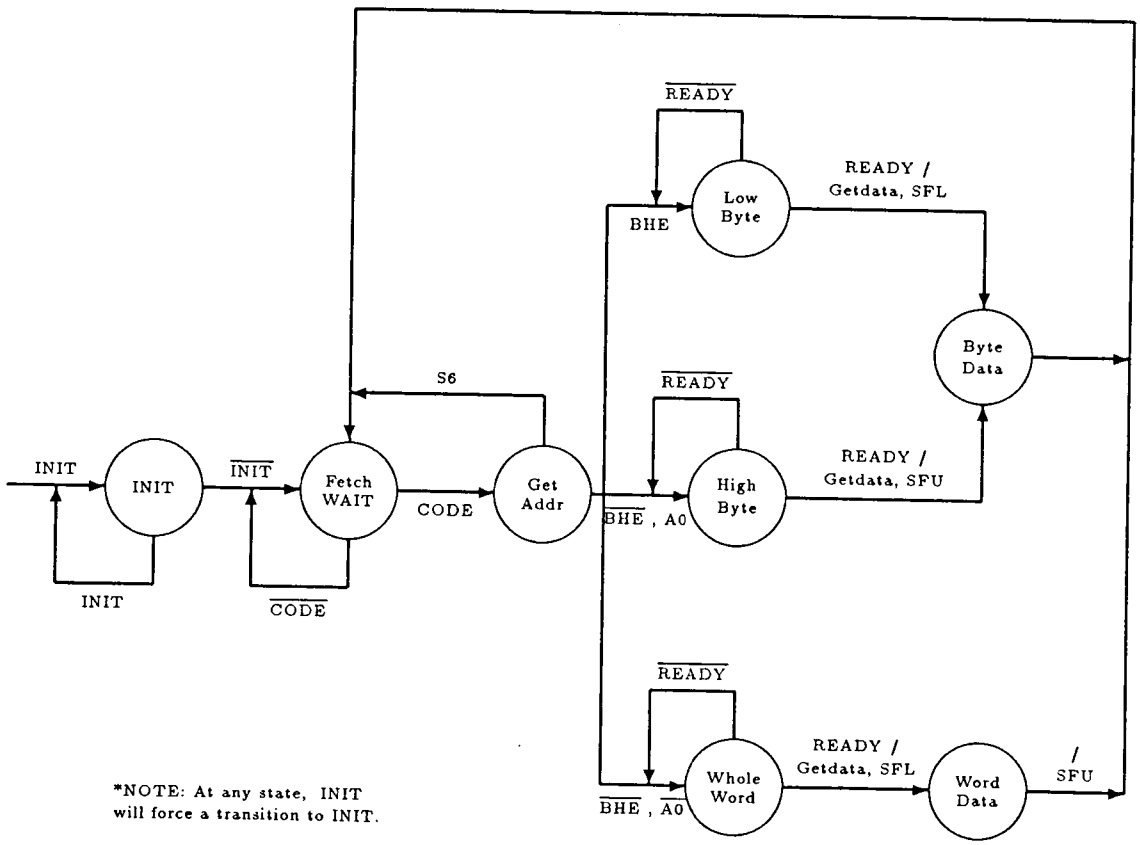


Figure 6.4: CODE FETCH FSM state diagram.

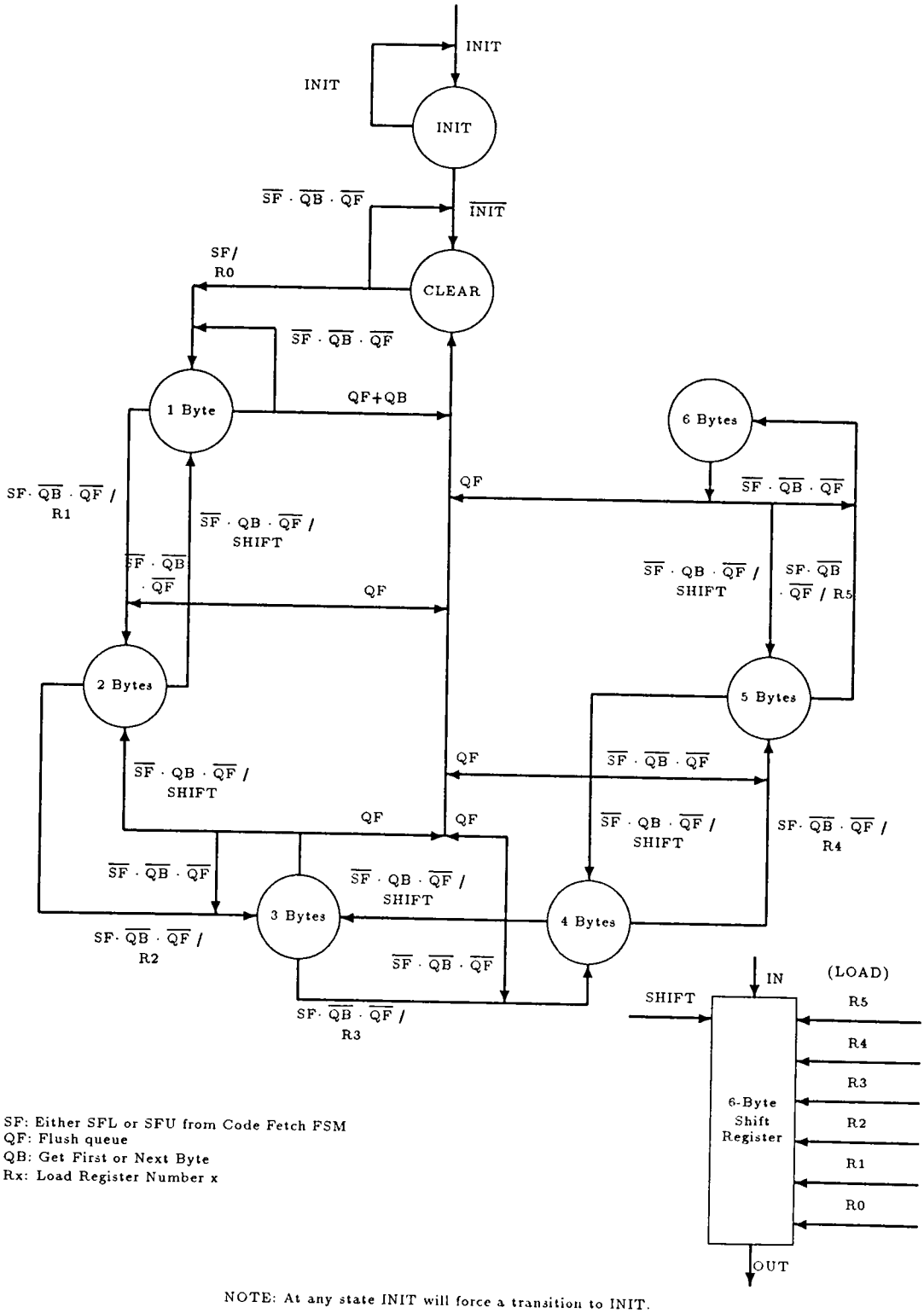


Figure 6.5: QUEUE CONTROL FSM state diagram.

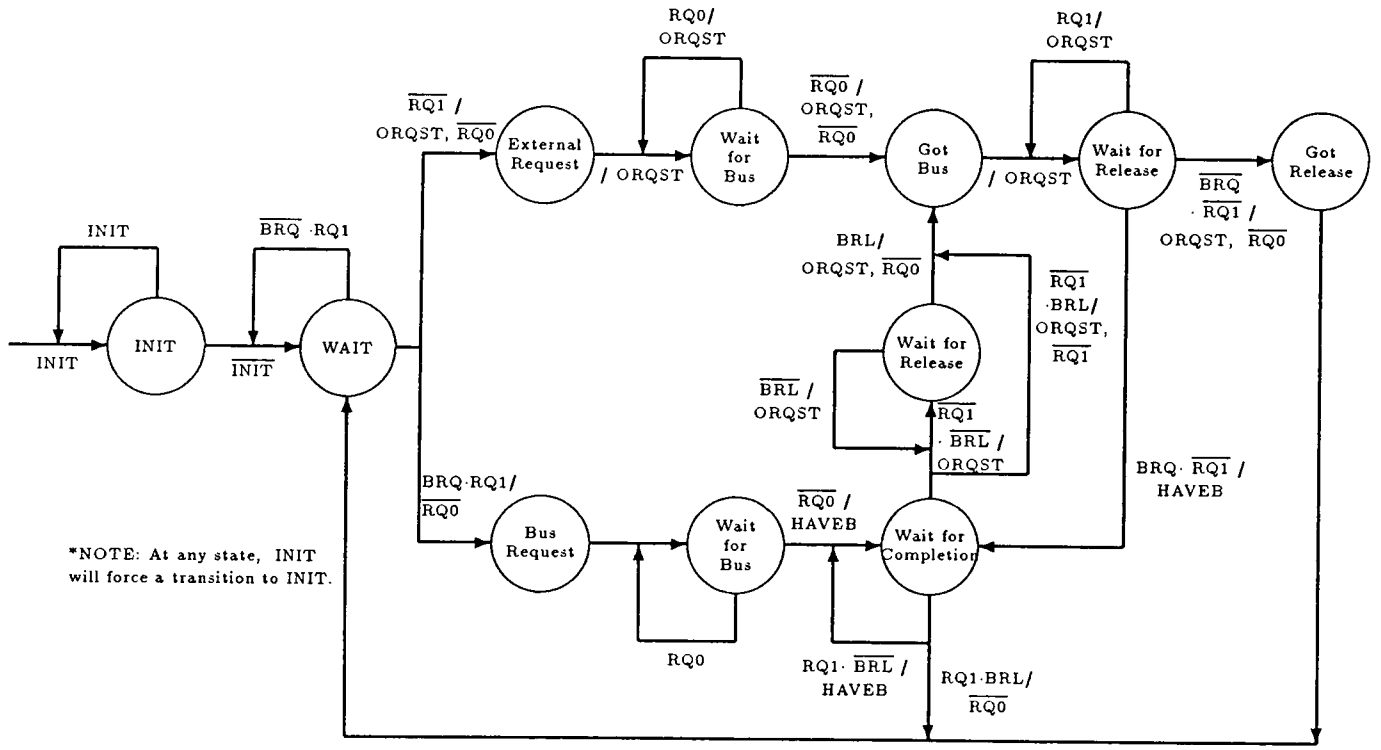


Figure 6.6: $\overline{RQ}$ / $\overline{GT}$ FSM state diagram.

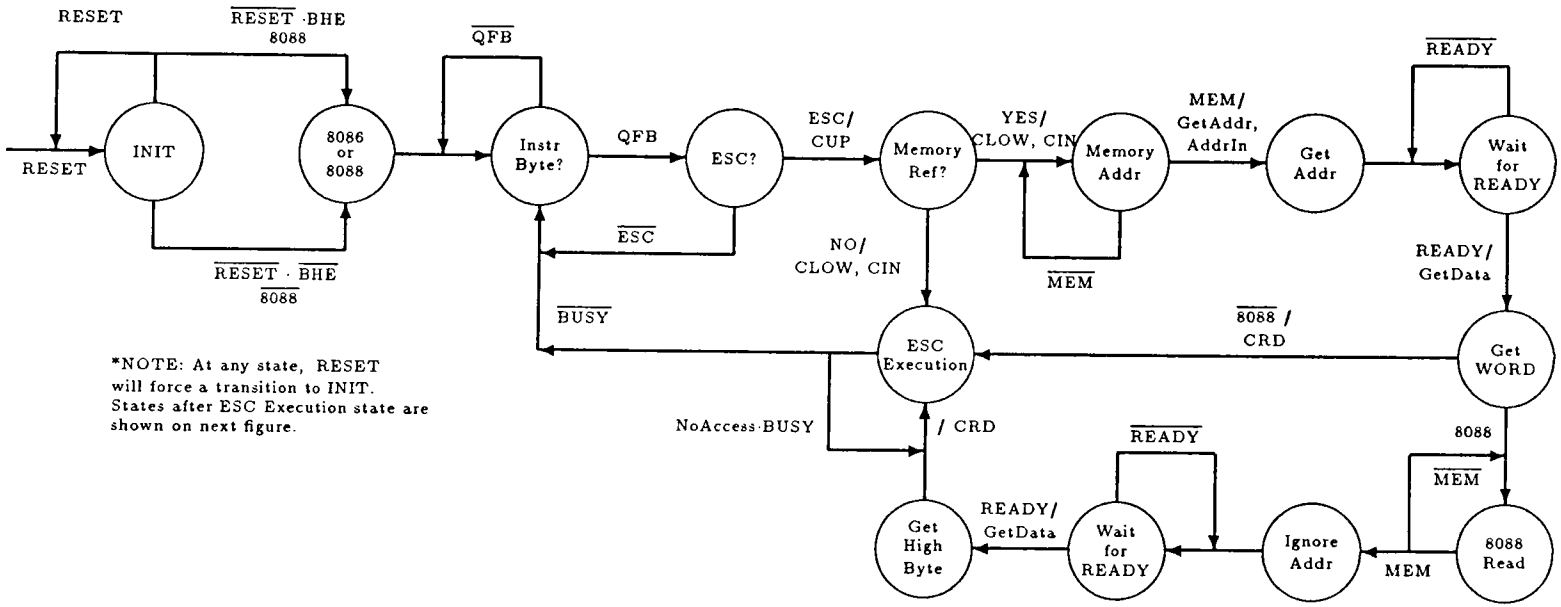


Figure 6.7: Interface FSM state diagram.

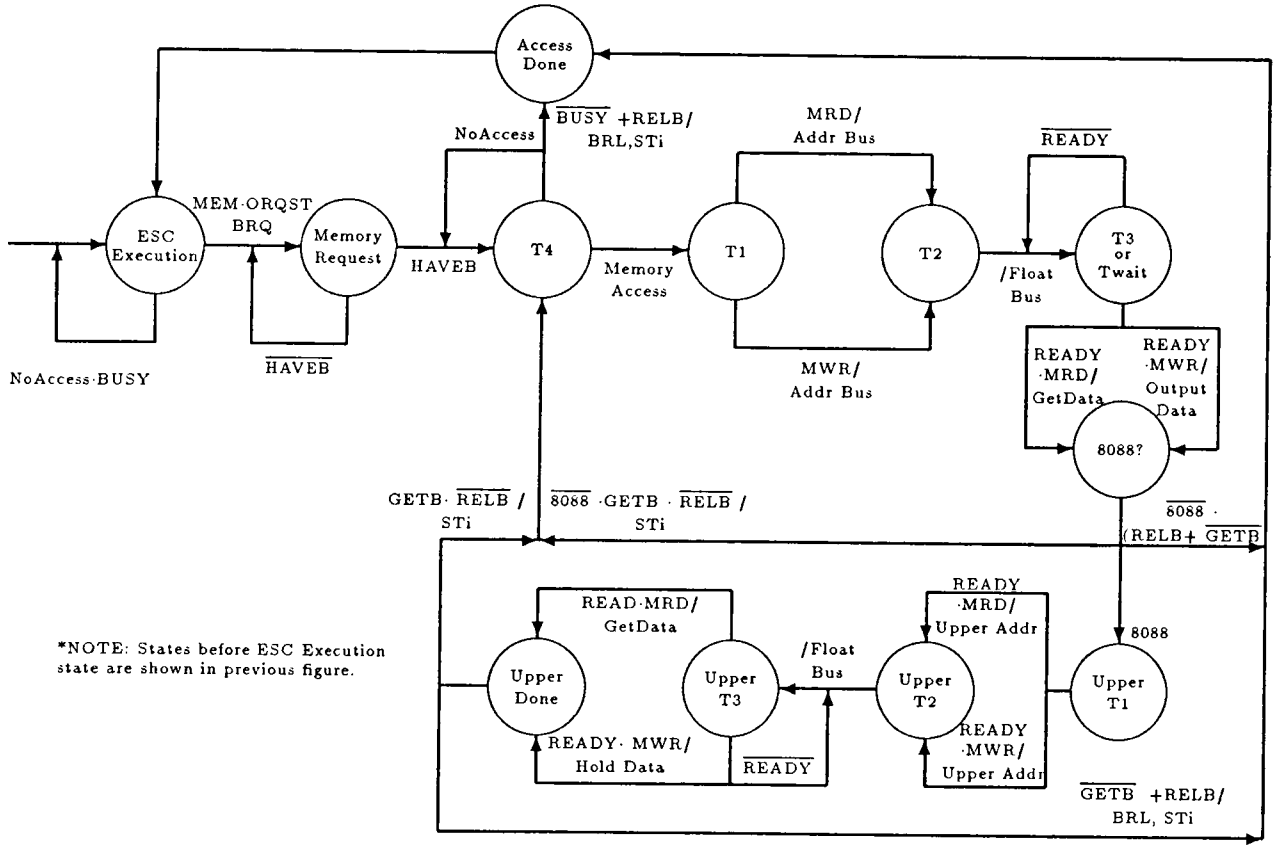


Figure 6.7(cont): Interface FSM state diagram.

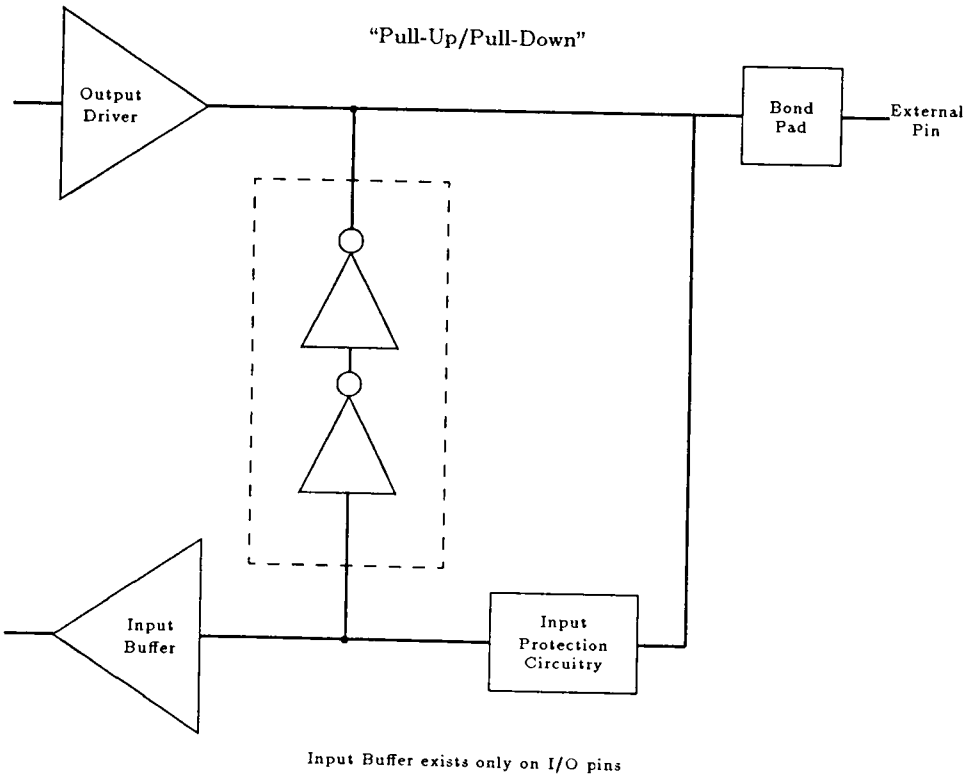


Figure 6.8: Bus-hold circuitry on *AD0-AD15*, *A16-A19*, and $\overline{BHE}$ / *S7* (After Intel [4] page 3-35).

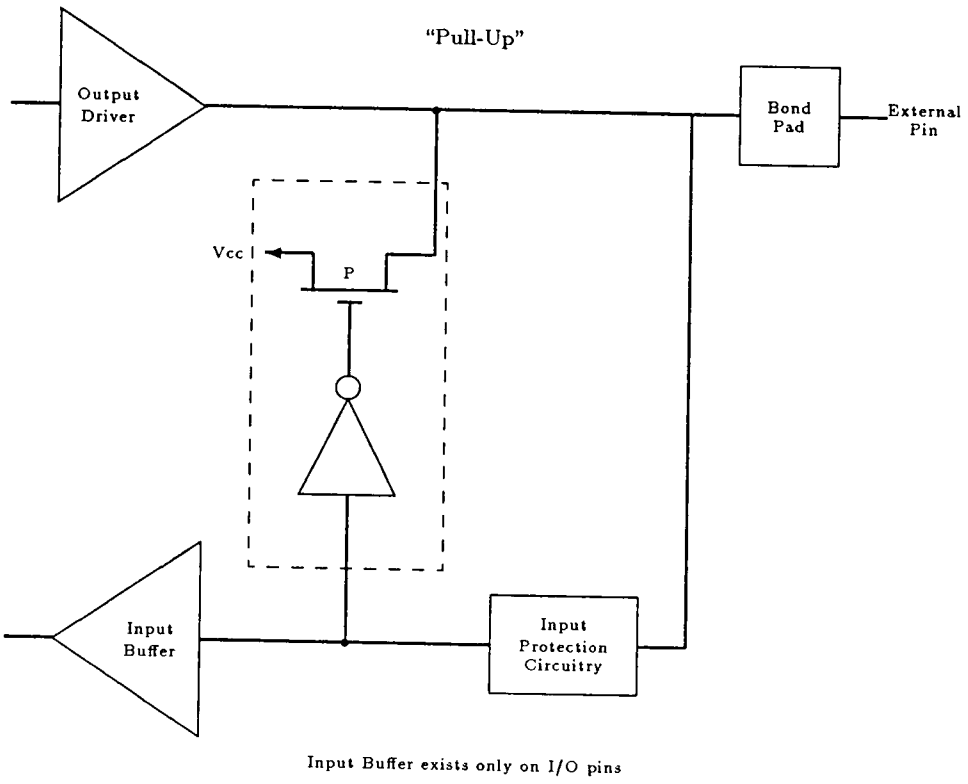


Figure 6.9: Bus-hold circuitry on $\overline{S0}$ - $\overline{S2}$ and $\overline{RQx}$ / $\overline{GTx}$.

This circuitry which is present in and was obtained from the CMOS version of the 8086 assumes a CMOS implementation and is used instead of pull-up transistors.

6.2 Software Considerations

The ESC instruction is used by the 8086/8088 to indicate a coprocessor instruction and has the same form as the ESC instruction for the 80286. When an ESC instruction is executed by the 8086/8088, I2 which has been tracking the processor execution captures the instruction and any address and data referenced by the processor for the instruction. The 8086/8088 can be synchronized with I2 by inserting a WAIT instruction either before or after an ESC instruction. The 8086/8088 tests $\overline{BUSY}$ when the WAIT instruction is encountered which indicates if I2 has completed the previous ESC instruction.

A sample use of the ESC instruction is:

```
ESC     InitCom
ESC     Oper,InAddr
WAIT
```

The first ESC instruction transfers the command InitCom to the custom circuit without transfering any address or data. This type of instruction would be used for such things as initialization, setup, or commands not requiring any data. The second ESC instruction transfers the command Oper1 to the custom circuit and captures the address InAddr and the data at that location and transfers those to the custom circuit. The custom

circuit via I2 can thus become the bus master and begin execution using data starting at address InAddr. By inserting the WAIT instruction after the ESC instruction, the 8086/8088 will not attempt any bus activity until after the custom circuit has indicated it is finished. Note that unlike with the 80286 the custom circuit can read and write data using the same ESC instruction in the 8086/8088 since I2 is performing the addressing.

6.3 CMOS Implementation Considerations

A CMOS implementation of I2 is relatively simple to perform using the Berkeley VLSI design tools and CMOS libraries available from MOSIS. The states of all the FSMs are known and a *peg* program can be written which will generate the equations. From these equations, *eqntott* is used to generate the truth tables which are used by PLA generators such as *mpla* to produce a CMOS layout of the FSMs. Thus, the FSMs are easily generated. The other circuitry is not as easily produced but does not require tremendous effort. The I/O and other pads are available from the CMOS libraries, while the remaining circuitry which includes nothing more than latches, dynamic and static registers, S-R flip-flops, and multiplexers can be produced by designing and laying-out a few simple cells.

The most difficult design problem for I2 is the clock. The above design assumes a two-phase non-overlapping clock. Due to the requirement that I2 be pin compatible with the 8087, only a single phase clock input is available. This single phase clock is the the main processor clock and all timing references are based on this clock. Thus, a two-phase non-overlapping clock must be derived from the single phase clock. Several designs are readily available which perform the derivation but the overall clock frequency is at least

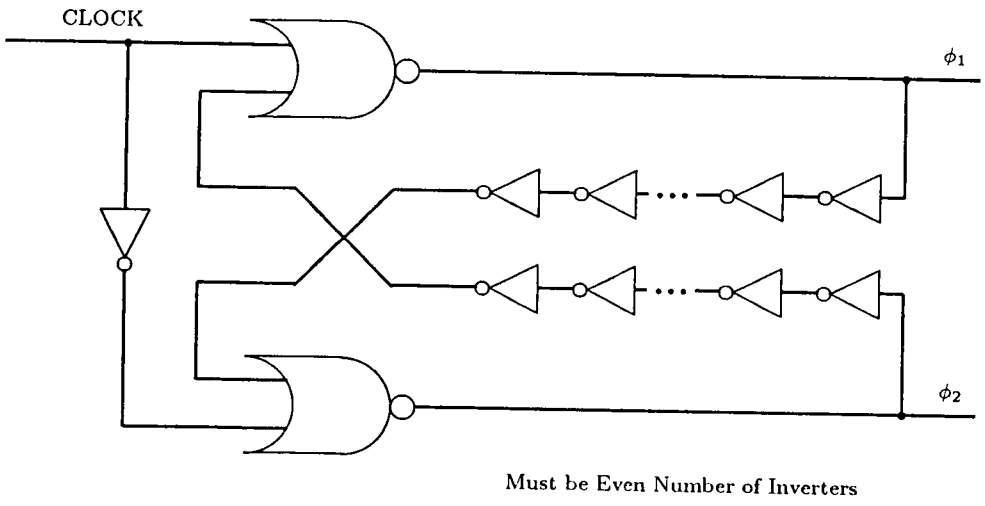


Figure 6.10: Non-overlapping clock circuit with return delay.

halved which is unacceptable. A workable alternative is to use a flip-flop arrangement with a long delay in the feedback path as shown in Figure 6.10. This design should work and does allow the user some degree of control in adjusting the overlap based on the length of the delay.

CHAPTER 7

SUMMARY AND CONCLUSIONS

A coprocessor design frame for use in VLSI circuit designs which is pin compatible with the Intel 8087 numeric coprocessor was designed and CMOS implementation considerations have been presented. Conceptual designs for coprocessor design frames for two other widely available microprocessors – the Motorola 68020 and the Intel 80286 – were developed. The conceptual design of the Motorola 68020 coprocessor design provides enough detail for a straightforward implementation if desired. On the other hand, an implementation of the Intel 80286 coprocessor design frame would require more detail concerning the execution of coprocessor instructions by the 80286.

Design frame constraints and a framework for internal protocols were presented which provide the basis for development of other design frames. All of the design frames would then be reasonably compatible and VLSI circuits designed for one design frame could easily be placed in another. Also as others have proposed, specialized design frames can be developed in areas such as image processing, pattern matching, artificial intelligence, and robotics[6,8,9]. This would significantly reduce the cycle time from idea to physical device.

Table 7.1 summarizes the design frames discussed in this work and the degree to which each meets the design frame constraints and other attributes. Each constraint or attribute adds greater usefulness to a design frame. A design frame that meets or exceeds the most constraints and has the better overall attributes will provide the better

Table 7.1: Design frame constraints and attributes.

CONSTRAINTS or ATTRIBUTES	DESIGN FRAMES						
	"BASIC"	MULTIBUS	8086	M1	M2	I1	I2
Conceptuality	Simple	Systems-oriented	Replace main processor	Co-exists as coprocessor	Co-exists as coprocessor	Co-exists as coprocessor	Co-exists as coprocessor
Universality	Limited	Any MULTIBUS system	Limited	68020-based systems	68020-based systems	80286-based systems	8086/8088-based systems
Robustness	Very	Plug-in MULTIBUS board	Pin compatible with 8086	Pin compatible with existing coprocessor	Pin compatible with existing coprocessor	Pin compatible with existing coprocessor	Pin compatible with existing coprocessor
Standardization	Standard package	MULTIBUS standard	8086 package	68881 package	68881 package	80287 package	8087 package
Systematics	Regular layout	Regular layout	PLA FSM structure	PLA FSM structure	PLA FSM structure	PLA FSM structure	PLA FSM structure
Geometry	Perimeter area	Perimeter area	Perimeter area	Block with perimeter area	Block with perimeter area	Block with perimeter area	Block with perimeter area
Electrical Specifications	Standard levels	Exotic circuitry	Standard levels	Standard levels	Standard levels	Standard levels	Standard levels
Assembly	Requires construction	Specialized board	Specific microcomputer board	Use existing coprocessor socket	Use existing coprocessor socket	Use existing coprocessor socket	Use existing coprocessor socket
Internal Protocol	Simple	Two-cycle handshaking	Four-cycle handshaking	Two-cycle handshaking	Two-cycle handshaking	Two-cycle handshaking	Two-cycle handshaking
Hardware Availability	Build-your-own	Specific	Specific	Number of 68020 systems on increase	Number of 68020 systems on increase	Number of 80286 systems on increase	Wide availability of 8086/8088 systems
Bus Access	N/A	Yes, master/slave	Yes, via MULTIBUS	Yes, via coprocessor instruction	Yes, via coprocessor instruction	Yes, via coprocessor instruction	Yes, direct bus control
Packaging	Standard MOSIS 40-pin	Standard MOSIS 84-pin grid array	Standard MOSIS 40-pin	Non-standard 68-pin grid array	Non-standard 68-pin grid array	Non-standard 40-pin	Standard MOSIS 40-pin
Software	N/A	User-written code	User-written code	Uses existing coprocessor instructions	Uses existing coprocessor instructions	Uses existing coprocessor instructions	Uses existing coprocessor instructions
Relative Speed	1	1	1	3	2	2	1

design frame for general use. But in certain specific areas such as image processing a specialized design frame is required which may not the best "general" design frame. This type of decision is best left up to the designer of the custom circuit and a table such as Table 7.1 should aid in that decision. The relative speed attribute indicates the penalty paid by using the design frame instead of directly interfacing the custom circuit to the outside world. The larger the number the greater the penalty.

As the table indicates, the "basic" design frame is very simple to use and does not have a large speed penalty but is limited in what type of custom circuits can be used with it. Any custom circuit which needs more than simple switches and LEDs should not be placed in the "basic" design frame. The MULTIBUS design frame is slightly more difficult to use but a broader range of custom circuits can be placed in it. The handicap with the MULTIBUS design frame is that a specific board which is not commercially available is required and code to access it from a MULTIBUS microcomputer system must be written by the user.

The 8086 design frame falls in the same category as the MULTIBUS design frame except that the microcomputer board in which it is installed is commercially available. The set of design frames described in this work, M1, M2, I1, and I2, are similar to the MULTIBUS design frame in ease of use but they have distinct advantages over it.

M1, M2, I1, and I2 co-exist as coprocessors in certain microcomputer systems and they all use existing coprocessor instructions within those microcomputer systems. The distinction between them is in what system they can be installed in, the chip packaging, bus control, and better speed. From the information available, I2, the Intel 8086/8088 coprocessor design frame, has the best combination of these. There are more 8086/8088 microcomputer systems available than the others, the 40-pin DIP package of the 8087 is

a standard package available from MOSIS, and the system bus can be directly controlled by the design frame. The better speed of I2 over the others is due to M1, M2, and I1 having all their data transfers handled by the main processor and not by themselves. Of particular note is that M1 must transfer an address and data to the main processor, the main processor accesses the memory specified, and then transfers the data back to M1 which is why it has a relative speed of 3.

By placing a custom VLSI circuit in the Intel 8086/8088 coprocessor design frame, a designer could use an existing system, such as the IBM-PC/XT or any of its clones, to test, evaluate, and make use of the custom circuit. If other coprocessor design frames are implemented, they could be used in the same manner in their corresponding microcomputer systems. All of which leads to the use of custom VLSI circuits by a much larger audience.

LIST OF REFERENCES

LIST OF REFERENCES

[1] *iAPX 286 - Hardware Reference Manual.*
Intel Corporation, Santa Clara, California, 1983.

[2] *iAPX 86/88, 186/188 User's Manual - Hardware Reference.*
Intel Corporation, Santa Clara, California, 1985.

[3] *MC68020 32-bit Microprocessor User's Manual.*
Motorola, Inc., Englewood Cliffs, New Jersey, second ed., 1985.

[4] *Microsystems Components Handbook, Microprocessors Volume 1.*
Intel Corporation, Santa Clara, California, 1986.

[5] N. A. Alexandridis, *Microprocessor System Design Concepts.*
Rockville, Maryland: Computer Science Press, 1984.

[6] A. G. Bell, "Using System Kits and Design Frames to Enhance VLSI Prototyping," Tech. Rep., Xerox Palo Alto Research Center, 1985.

[7] G. Borriello, R. H. Katz, and A. G. Bell, "The Multibus Design Frame," Tech. Rep. UCB/CSD 85/232, Computer Science Division (EECS), University of California, Berkeley, California, May 1985.

[8] G. Borriello, R. H. Katz, A. G. Bell, and L. Conway, "VLSI system design by the numbers," *IEEE Spectrum*, pp. 44–50, February 1985.

[9] L. Conway and A. G. Bell, "System Kits, Design Frames, and Network Services for the Rapid Prototyping of Advanced Computing Systems," Tech. Rep., Xerox Palo Alto Research Center, 1983.

[10] G. W. Gorsline, *16-bit Modern Microcomputers - The Intel I8086 Family.*
Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1985.

[11] R. H. Katz, S. Weiss, and R. Kelkar, "A Experimental Design Frame for VLSI Circuit Prototyping," *VLSI Design*, pp. 57–58, May/June 1982.

[12] Y. Liu and G. A. Gibson, *Microcomputer Systems: The 8086/8088 Family.*
Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1984.

[13] J. Newkirk and R. Mathews, *The VLSI Designer's Library.*
Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1983.

[14] A. Paeth, "The Design of a Basic Design Frame," Tech. Rep. Memo KB-VLSI-83-1, Xerox Palo Alto Research Center, 1983.

[15] The MOSIS Project, "The MOSIS System (what it is and how to use it)," Tech. Rep. ISI/TM-84-128, Information Sciences Institute (ISI), University of Southern California, Marina Del Rey, California, 1984.

VITA

Danny Fieldon Newport was born in LaFollette, Tennessee on August 14, 1957. After graduating in June 1975 as valedictorian from Jacksboro High School in Jacksboro, Tennessee, and a brief appearance at the United States Military Academy at West Point that summer, he enrolled in the College of Engineering at the Knoxville campus of the University of Tennessee. While at the University, he participated in the Cooperative Engineering program and worked for eight quarters in the Electronics Technology group in the Development Division of the Oak Ridge Y-12 Plant. He was graduated with a Bachelor of Science Degree in Electrical Engineering with high honors in August 1981.

After receiving his B.S., he began work as a Development Associate with the same group he had co-oped with at the Y-12 Plant which was at that time operated by Union Carbide Corporation. While at Y-12, he enrolled in the graduate program in Electrical Engineering at the University of Tennessee in March 1982.

In March 1984, he left Union Carbide and began working for the University of Tennessee as the systems manager for the College of Engineering Computer Laboratory. In this position, he has been responsible for the installation, maintenance, and operation of a one-half million dollar computer facility devoted to faculty and graduate research.

He has been married to Rosemary Warren Newport since 1985 and they share a house with their cat, Warlock.