1985

Using N.2 to Model a Microprocessor System

Benjamin J. Patz
University of Central Florida

USING N.2 TO MODEL A MICROPROCESSOR SYSTEM

BY

BENJAMIN JOSEPH PATZ
B.S., Rensselaer Polytechnic Institute, 1983

RESEARCH REPORT

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in the Graduate Studies Program of the College of Engineering
University of Central Florida
Orlando, Florida

Summer Term
1985

ABSTRACT

Due to the complexity of designing digital systems using VLSI parts, a tool for aiding in system level design specification and verification is needed. Functional level modeling languages and simulators provide that tool. An example of such a tool is the N.2 package of software produced by Endot Inc. and soon to be running on a VAX computer at the University of Central Florida.

An overview of the N.2 system is presented in this paper with emphasis on the modeling language of N.2, ISP'. A Small Instruction set Computer (SIC), originally specified in AHPL, is designed with this software using several design methodologies. These range from an instruction level implementation to a microcoded register level implementation. The ISP' source code is provided for each implementation.

Comments on the ability of the N.2 software to model systems at various levels of design abstraction are made. A comparison of the functional modeling language of N.2, ISP' to other functional level design languages is made. Finally, some areas that warrant further investigation are presented.

ACKNOWLEDGEMENTS

The author would like to express his appreciation to those whose encouragement helped see him through the completion of this paper. Those individuals include the author's mother, Anna Mae Patz, and father, Dr. Benjamin W. Patz. In addition, the author would like to thank the Technical Computing Center of Martin Marietta Aerospace in Orlando, Florida, for providing the computer resources on which this paper was created.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SECTION 1
INTRODUCTION

Due to the complexity of designing digital systems using VLSI parts, a design specification and verification tool is needed. Traditional hardware support tools, such as gate level modeling software, do not provide adequate capabilities early in the design cycle. A functional level modeling tool is essential for this task. The N.2 system (Ordy 1983) includes such a tool. In this section a description of functional level modeling is presented. The N.2 system is then discussed. Finally, a brief introduction to a Small Instruction Set Computer (Hill 1978), which was modeled with the N.2 software, is presented.

Functional Design Modeling

A traditional breakdown of the hierarchy of digital design is given in Figure 1. The five levels of modeling are described in more detail below.

Behavioral level models are those models that a system level designer would use to describe the general function to be performed in a given design. There is little or no relationship between this model and the hardware that is used to implement it. The behavioral model

is simply a discussion of the behavior of the overall system with respect to the performance of a certain task.

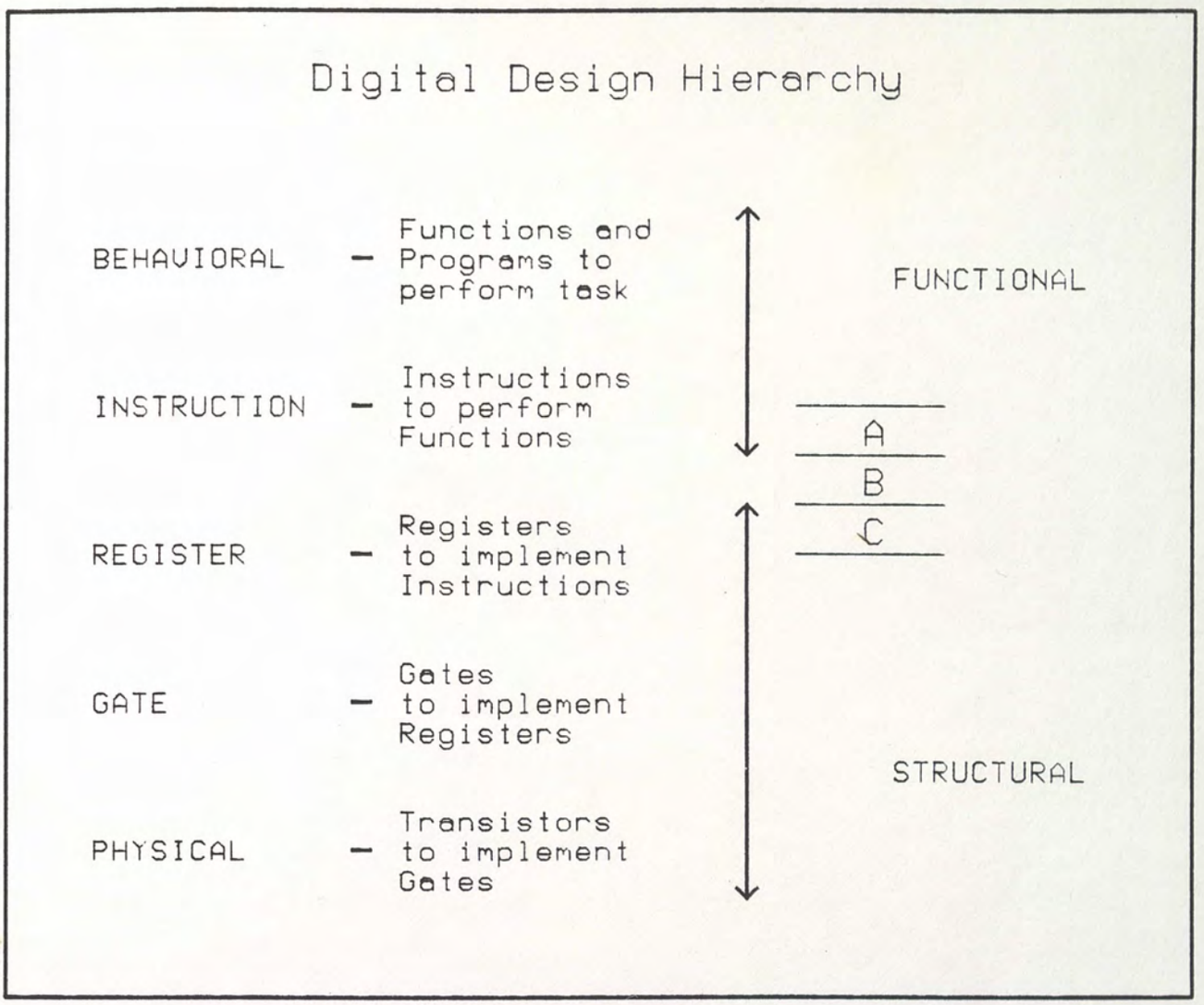

Figure 1. Digital Design Hierarchy.

In order to begin to implement a behavioral level model a set of primitive functions, or instructions, is developed which can be combined to perform the more complex functions necessary to solve a task. Instruction level models are models that execute this set of instructions,

and define a machine similar to what an assembly level programmer would see. The purpose of the instruction level model is two-fold. First, early in the design cycle, it provides a method of examining the completeness and performance of an instruction set. Later in the design cycle it can act as the core of a software emulation system for the development of software for a target machine.

Where the instruction level model defines the instructions necessary to perform functions, the register level model defines the physical hardware blocks necessary to perform the instructions. These models begin to define the physical structure of a machine in the form of memories, registers, busses, and combinatorial function blocks. Memories and registers are considered functional primitives, and combinatorial blocks are treated as functions that execute with ideal zero delay. Registers and memories contain rudimentary timing information in the form of clock periods necessary to perform their respective functions. The intent of the model is to examine these internal structures of the machine.

An obvious extension of register level models includes more accurate timing information. Gate level models provide for this extension. At this level of modeling the performance of a macroscopic function is almost lost, and the concentration is on the performance of microscopic functions. The primitive elements are gates, which are combined to form registers and other structures. These gates contain

timing information in the form of input-to-output propagation delays, which can be nominal or worst case values.

At the lowest level of modeling is physical modeling. Here timing is the primary concern, usually in the form of timing variation over temperature and variation of processing parameters. Transistors are the primitive elements. Due to the complexity of models at this level, typically only a very small portion of the original design is modeled at one time.

It is convenient to map these five levels of design abstraction into only two levels: functional, and structural. These two levels answer the two fundamental questions of digital design "What is done" and "How is it done." This division is convenient also because current software tools fall fairly neatly into one or the other of these classes. For example, the N.2 software has its primary use at the functional level. Simulators like CADAT* perform best at the structural level.

The purpose of this paper is to examine design at the lower reaches of the functional level and the upper reaches of the structural level. Three divisions are made in the design hierarchy. These divisions are labeled simply A, B, and C. The capability of the N.2 software to design at each of these three levels is discussed.

---

*CADAT is a logic simulator marketed by HHB Softron Inc.

N.2 Software Environment

The N.2 design environment contains six major components which work together to produce a model of a multi-processor system. A simplified block diagram of these parts is shown in Figure 2. Since the intent of this paper was not to examine the syntax or the detailed capabilities of each of the N.2 modules, only enough explaination is provided to allow for a general understanding of the capabilities of N.2. Particular emphasis is placed on the hardware modeling language of N.2, ISP', and on the the tool for modeling instruction sets, metaMicro.

ISP'

ISP' is the functional modeling language of the N.2 system. Its purpose is to allow the designer to create a source code for various hardware designs. These designs are later combined and simulated.

Hardware designs in ISP' are essentially a collection of processors that are connected together through ports to form a network. Each processor type is defined in its own ISP' source file. Multiple instances of these processors, each using the same ISP' file definitions, may be placed in a network. The instantiation of these individual processors and their interconnections describe the topology of the network.

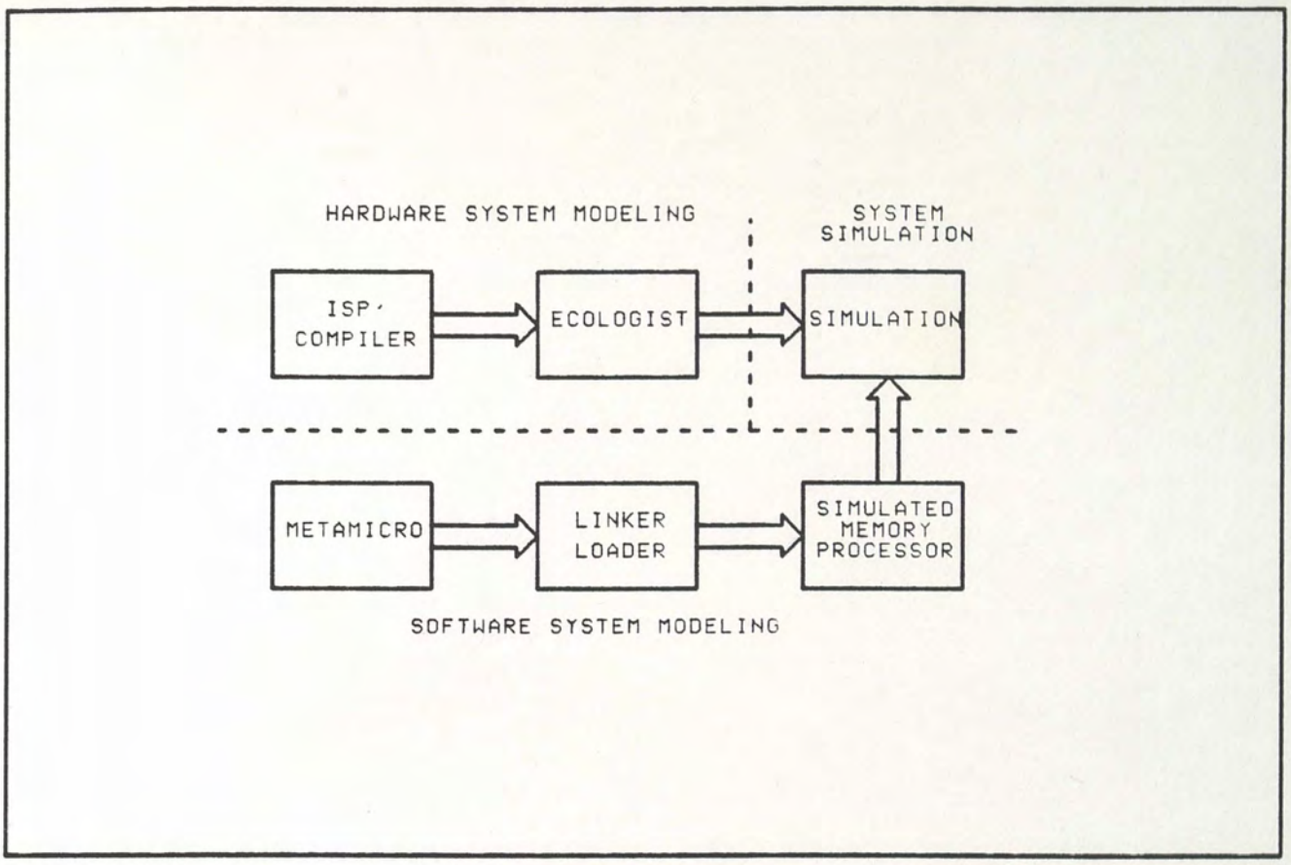

Figure 2. N.2 Block Diagram.

A processor, as viewed in the ISP' language, is a collection of processes. Functionally, processes are defined as a collection of procedures, functions, and commands. Structurally, processes are defined in the declaration statements of the ISP' code. The basic constructs are defined here in order to better understand any ISP' implementation methodology, and in particular the implementation of the Small Instruction Set Computer.

The two types of processes are the main and the when process. A single main process may exist in any ISP' processor. This process repeats itself continually, restarting itself as soon as it terminates. This type of process is useful if a cyclic function; such as instruction fetch, instruction execution, instruction fetch, ... ; is being performed by a processor. The when process, on the other hand, is only activated when a particular condition occurs on a port. These conditions are either the detection of a low to high transition (lead) or high to low transition (trail) of a particular signal. Because of the when process, multiple operations may be occurring at the same time.

The following example demonstrates a process. This particular process performs a read from memory. It is activated when the CLOCK rises, and the READ line is high. Data is then transferred from memory, at the point where the ADDR is currently pointing, and placed on the port DATA.

```
when (clock : lead (read eql 1)) :=
(
 data = M[addr]
)
```

In either case, main or when, each process is a collection of ISP' statements and procedure and function calls. Procedures and functions are defined in an analogous manner to procedures and functions in a

higher level language like Pascal. The ISP' language is therefore procedural in nature, yet, due to the when processes, it is also parallel. The following procedure could be used by a processor to retrieve a word from memory, if the memory possessed an address and a data register.

```
get_word :=
(
 MD = M[MA]
)
```

The structure of a processor is defined by the declaration section of the ISP' code. There are three major declaration items in ISP'. States represent registers or latches. Ports are pins that form connections to the external world, and allow for interprocess communication. Macros and formats are ways of giving symbolic names to logical entities which may represent only part of a physical entity. They are also used for improving the readability and structure of the resulting ISP' code.

The following ISP' code demonstrates the declaration of a few of the facilities of a computer. First, a word length is defined. Second, several registers are defined. Finally these registers are formatted so as to make the extraction of the opcode and address easier.

```
macro   WORD = 18 &,
        ADDRESS = 13 &;

state    IR <WORD>,
         AC <WORD>,
         PC <ADDRESS> ;

format  opcode = IR<17:15>,
        addr   = IR<12:0>;
```

When an ISP' simulation is running it is a collection of cooperating and competing ISP' processes. The runtime kernel controls the scheduling of events and invocation of processes in the simulation. Events are of two types, a port changing value, and a scheduled wakeup call that occurs after a processor issues a delay command. The delay command is the only command in ISP' that can be used to carry timing information. This delay is defined in terms of user time units, which are specified at the time the processor is instantiated in the network. The following example illustrates the use of the delay statement.

```
when (ck : lead (read eql 1)) :=
(
 delay(3);          ! wait three cycles
 MD = M[MA];        ! get data
)
```

As opposed to a sequential language, which performs operations one after another, ISP' performs operations in parallel. There are cases, however, in which a user would wish to force the order of operations. This is done with the use of the special command NEXT. All operations up to a next are performed simultaneously, and before all the operations following the NEXT. Certain ISP' commands imply a NEXT statement. For example, a NEXT is implied before a delay or wait statement, at the end of a process, and at the end of a procedure or function. The example below illustrates the simultaneaity of two operations. The PC register is incremented at the same time that it is transferred to MA. Thus if prior to the call to get_word, PC held the value 5, then after get_word was executed, data would be fetched from memory location 5, and PC would have the value 6.

```
get_word :=
(
 PC = PC + 1;          ! inc PC
 MA = PC; next;        ! transfer PC to MA
 MD = M[MA]            ! get data
)
```

Arithmetic and logical operations in ISP' have been designed to reflect operations in real ALUs. All operations operate on operands of specific width and produce a result of a specific width. To store results in a wider storage area leading 1's or 0's must be appended to the

structure. This is accomplished with the SXT (sign extend) and EXT (zero extend) operators. Arithmetic operations are performed with 2's complement arithmetic. Assuming X, Y and Z are 8, 8 and 16 bits respectively, and X and Y both contain 1A hex, the following ISP' statements leave Z with the values 34 and F4 respectively.

```
Z = (X + Y) ext 16;
Z = (X + Y) sxt 16;
```

ISP' allows the user to insert comments throughout his code. A commenting convention has been adopted throughout this paper. At the beginning of the code a brief description of the processor is given, its current version, and a list of any references that would be helpful in understanding its function. Each declaration is described following the description. A description of every procedure or function is given. In large blocks of code, comments are inserted to facilitate the understanding of the overall function of that block. The goal of the comments is to provide a top down understanding of the code without providing excessive detail except in extraordinary situations.

The ISP' compiler translates the ISP' source code into an object file. These object files are linked together by the Ecologist to form a simulation program.

Ecologist

The combining of ISP' output files is controlled by the designer through the topology file. The purpose of the topology file is to resolve the ISP' references to ports, memories, and time, and to define the interconnection of multiple ISP' processors. Five different sections form the topology file.

The ports of one processor are connected to the ports of another processor by signals. If an ISP' model uses ports then there must be a signal declaration in the topology file. During simulation, the value of a signal is the logical OR of all ports tied to it. Each topology file has only one signal declaration section.

A collection of processor definitions follows the signal declarations. Each processor definition references an ISP' output file, and more than one processor can reference the same ISP' output file. Thus, multiple instantiations of a particular ISP' model can be made.

If the ISP' model makes a delay call then a time delay declaration is required for the model. The time delay declaration is used to give the relative time delays specified in the ISP' source file a physical meaning.

The connection of the ports of a given model to signals is defined in the connection declaration of each processor. Not all ports of a model must be connected to signals.

Finally, the initial contents of all ISP' memories is determined by assigning the ISP' memory name of each model to a memory file. These files may be created by the Linker/Loader, and could contain code for a processor to execute. A tool for aiding in the generation of this code is the metaMicro program.

metaMicro and the Linker/Loader

The metaMicro is a microassembler which utilizes a description of a processor's instruction set to assemble programs. The Linker/Loader is used to allocate this code into program memories. The metaMicro program consists of two major sections.

A declaration section allows a processor to be defined for code generation. Included in this section is the instruction length declaration and the format of the instruction or microinstruction word. An extended macro definition capability can then be used to define mnemonics for the instructions.

Following the declaration section is the instruction section. This section contains the instructions that are to be assembled for the target machine. For most machines these will be simply a list of macros that have been defined in the declaration section.

The Linker/Loader provides a generalized address resolution system which supports relocation of code. The designer also defines the allowable memory space for code generation in the Linker/Loader.

Simulated Memory Processor

The simulated memory processor prepares the list of memories referenced by the Ecologist and the memory contents created with metaMicro for simulation. All of the Linker/Loader output files are converted from their packed format to a page format that the simulation program can use. The simulated memory processor also produces a symbol table file containing the name of the memory files available to the simulation.

Simulation

Combining the output of the Ecologist and the Simulation Memory Processor yields the executable simulation of the processor system. A runtime kernel controls the execution of the simulation, and allows for user intervention. The goals of the simulation include functional verification of the design, and perhaps some timing analysis. The particular goals depend on the level of design and on the system being designed.

A Small Instruction Set Computer

A Small Instruction set Computer (SIC) was modeled using the N.2 software. This computer was chosen for two reasons. First, it is fairly simple yet displays most of the common features of computers.

Second, it is the computer used in the University of Central Florida's Computer System Design course, as a model computer.

The SIC machine is a 6 register computer. A brief description of the registers is given below.

```
IR - instruction register  18 bits
MD - data register         18 bits
AC - accumulator           18 bits
PC - program counter       13 bits
IA - index register A      13 bits
IB - index register B      13 bits
```

The word length of the machine is an unusual 18 bits, 13 of which can be used for addressing memory. Any of 8192 words of memory can be referenced with one of four addressing modes. The addressing modes are given below.

```
DIRECT     - effective address = address part of IR
INDIRECT   - effective address = address pointed to by
                                 the address part of IR
INDEX A    - effective address = address part of IR + IA
INDEX B    - effective address = address part of IR + IB
```

The instruction set of the machine includes instructions that reference memory (MRI), instructions that perform various operations (OPERATE), several input/output instructions (IO), and interrupt instructions (INT). These instructions are described in more detail in the next section. An AHPL description of the machine is given in Appendix A. It is appropriate to point out the difficulty in reading the

AHPL code and determining the function it is trying to perform. It will become apparent that ISP' code is much easier to understand.

The shortcomings of the SIC machine include a lack of regular structure, for example there are two separate register sizes. Also only one register, the accumulator, can be used for the arithmetic operations. The machine is complete enough, however, to exercise many of the capabilities of the N.2 software.

SECTION II

SIC IMPLEMENTATIONS

Overview

In order to demonstrate the capabilities of N.2 in modeling at various levels of design abstraction, three models of SIC are constructed, corresponding to the three classes of the design hierarchy presented in the Section I. These models are roughly analogous to those used by Motorola (Druian 1983) in developing the MC68000. The different model types are denoted by the three letters A, B, and C.

The Class A model, the most primitive of the three, is a model of SIC at the instruction level. Little attention is paid to timing information, and no external interface capability is provided. The purpose of this type of implementation is to provide a simulation model that can be used to exercise the instruction set of a particular machine in order to examine its richness relative to a target function (e.g., general purpose computing, signal processing, etc.).

An obvious extension of the Class A model is to model internal operations and external interactions in order to increase the detail of the timing information. The Class B model accomplishes this by providing a pin level model of the SIC machine. The instruction

execution is broken down in terms of register transfers. Each transfer is assumed to take one clock cycle. To provide for external interactions, the I/O facilities of SIC are also implemented and a sample I/O model is presented.

The Class C model is developed to further define the inner workings of the SIC machine. The register transfer control logic is implemented using a microcontroller. Each facility that could be controlled by this microcontroller is developed as a separate ISP' module. In much the same manner that a breadboard of a design would be constructed from off-the-shelf parts, the SIC machine is constructed from these many modules.

Modeling of each SIC model follows the design flow shown in Figure 3. A facility specification, functional flow chart, and instruction set description are presented for each model. The facility specification is used to develop the declaration section of the ISP' code and the interconnection topology for the model. The functional flowchart determines the overall flow for the ISP' code. The instruction specification helps determine the functions and procedures of the ISP' code as well as aiding in the metaMicro creation of the instruction models.

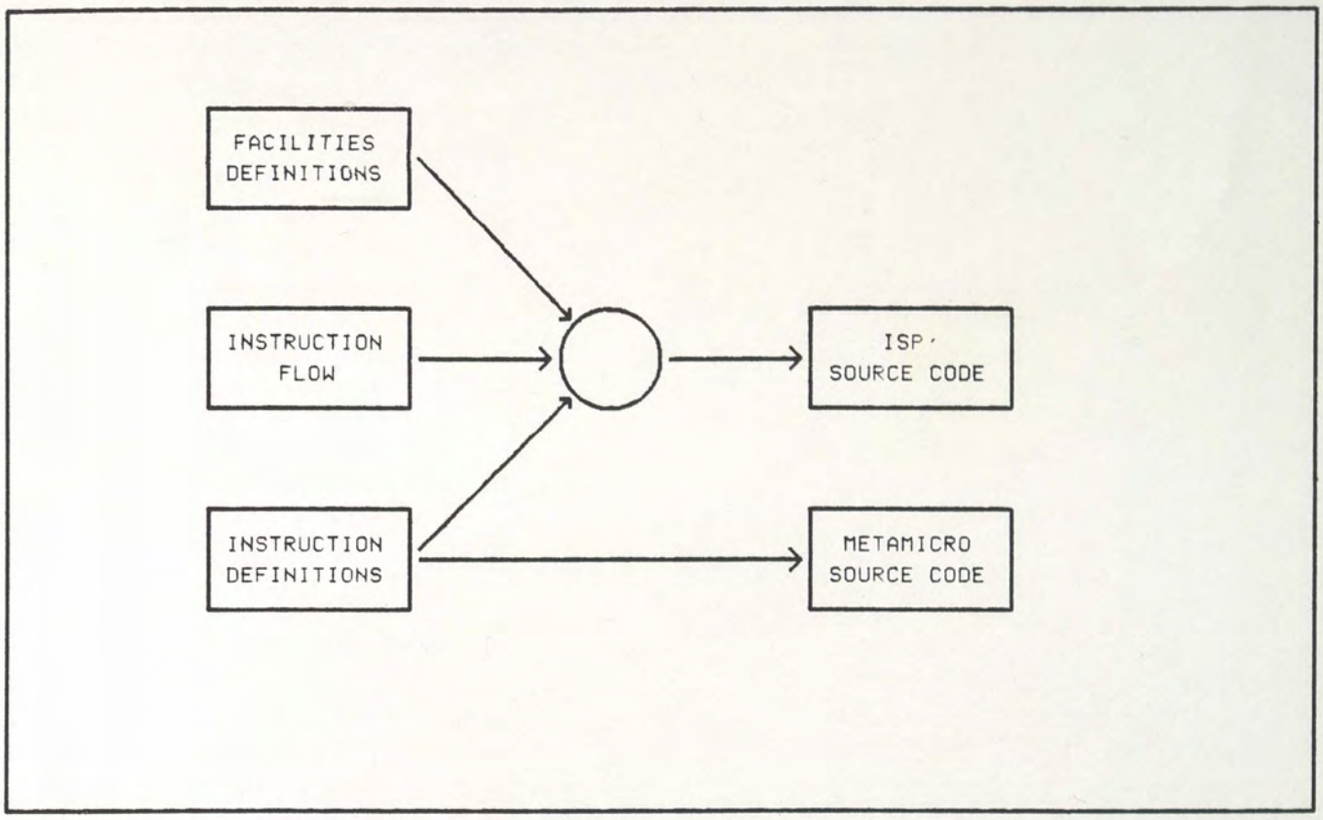

Figure 3. Model Development Flow.

Each model is discussed below. Some comments on the design process are provided in order to aid future designers in designing N.2 models. Also a description of the salient features of each model is provided.

Class A Implementation of SIC

The facilities set implemented in the Class A SIC is shown in Figure 4. This includes the major internal registers, the accumulator, AC, the memory data register, MD, etc. All the hardware to implement

the MRI and OPERATE instructions is included. These facilities define the declaration section of the ISP' code for the model. This is shown below. Since several of the registers are the same size, either 18 or 13 bits, constants are used to reference register sizes. This improves program readability, and also improves flexibility.

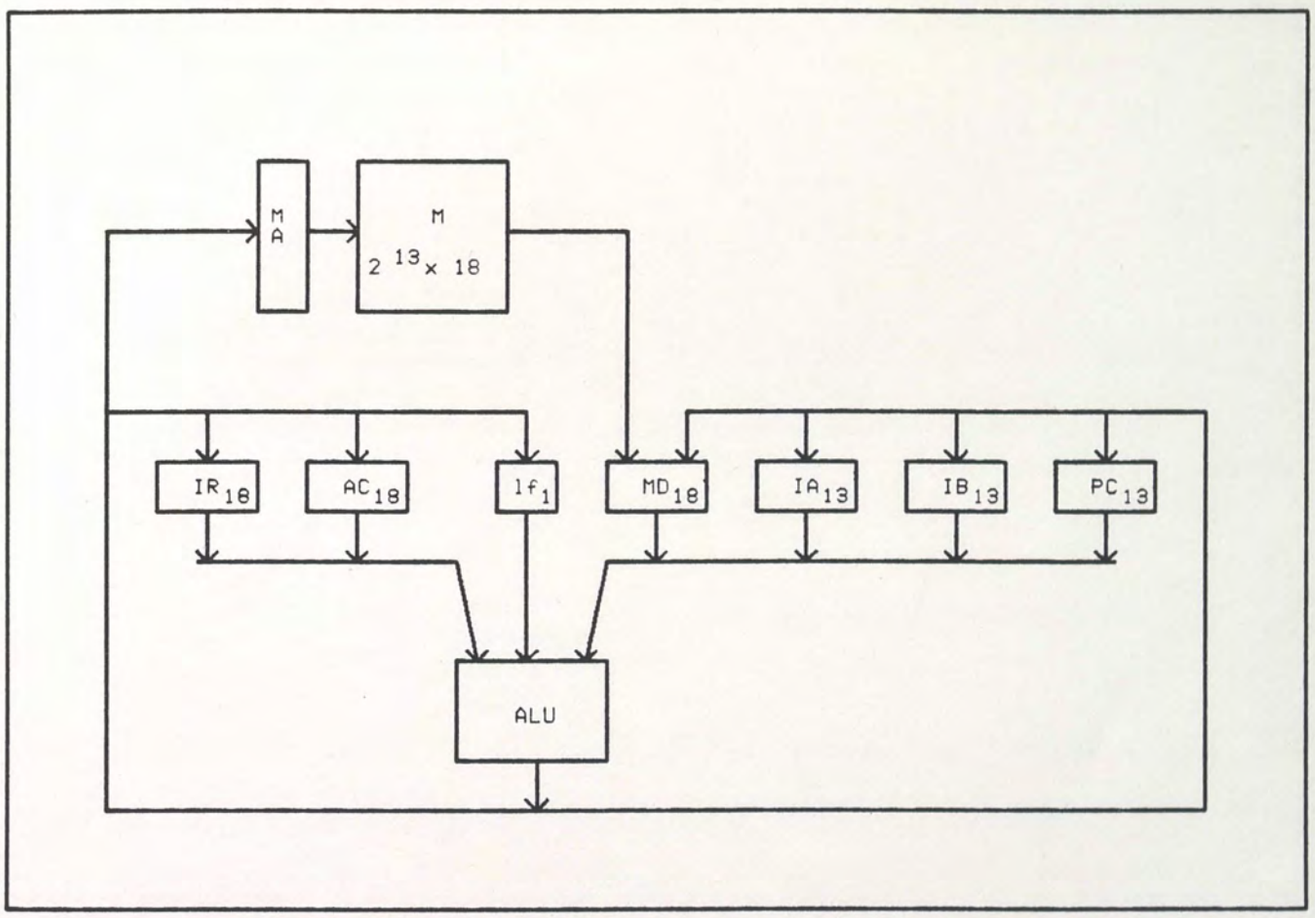

Figure 4. Class A Sic Facilities.

```
state    AC    <WORD>,          ! accumulator
         MD    <WORD>,          ! memory data register
         IR    <WORD>,          ! instruction register
         PC    <ADDRESS>,       ! program counter
         MA    <ADDRESS>,       ! memory address register
         IA    <ADDRESS>,       ! index register A
         IB    <ADDRESS>,       ! index register B
```

```
        lf;                             ! link flag

memory  M [0:8191] <WORD>;              ! program memory

format  opcode      = IR<17:15>,        ! operation
        addr_type   = IR<14:13>,        ! address type
        addr        = IR<12:0>,         ! address
        op_part     = IR<13:0>;         ! operate part of instruction
```

From the instruction format shown in Figure 5 and the instruction definitions of in Table I, the instruction register IR can be formatted. This allows specific bit fields, such as the opcode field, to be referenced as primitives. This also improves the readability of the code. A flowchart for an instruction cycle is shown in Figure 6. The basic flow follows the following steps : instruction fetch, instruction type determination, instruction execution. Since SICs flow is basically a cyclic process, the main block of code is coded in an ISP' main process. Each instruction described in Table I is coded as a separate procedure. This allows easier debugging of code, as well as improved flexibility. For example, a delay associated with the execution of each instruction can now be included in each of these procedures. An example of the code is given below, while a complete listing is provided in Appendix B.

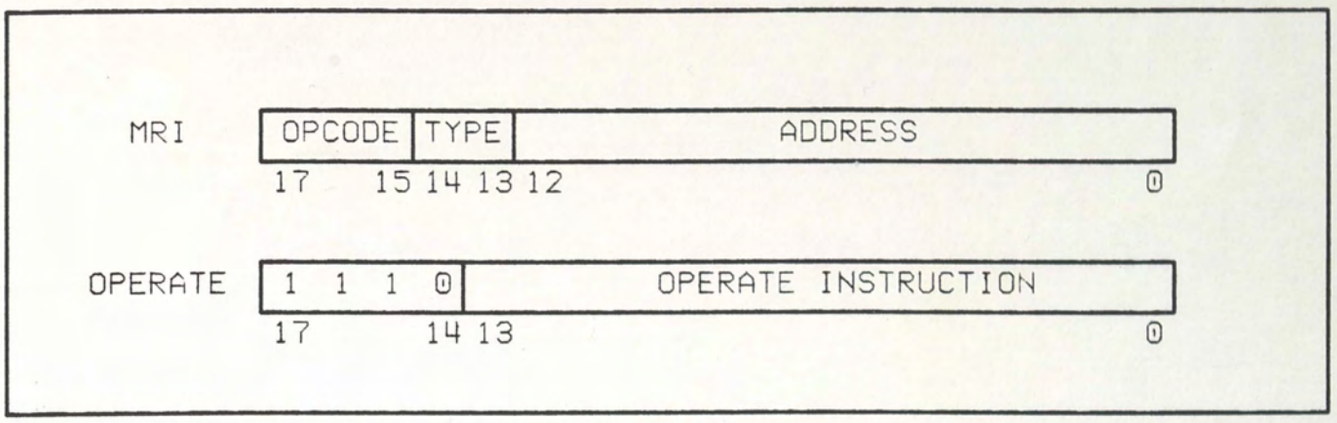

Figure 5. Class A Sic Instruction Format.

TABLE I

CLASS A SIC INSTRUCTION SET

OPCODE	MEANING
ISZ	increment memory and skip if zero
LAC	load accumulator
AND	AND memory with accumulator
TAD	twos complement add
JMS	jump to subroutine
DAC	deposit accumulator
JMP	jump
HLT	halt
NOP	no operation
CLA	clear accumulator
STA	store accumulator
CMA	comEplement accumulator
CLL	clear link
STL	set link
SKP	skip if accumulator >= 0
SKZ	skip if accumulator = 0
SZL	skip if link = 0
RAR	rotate accumulator right
RAL	rotate accumulator left
DTA	deposit accumulator in IA
DTB	deposit accumulator in IB
DFA	deposit IA in accumulator
DFB	deposit IB in accumulator
INA	increment IA
INB	increment IB

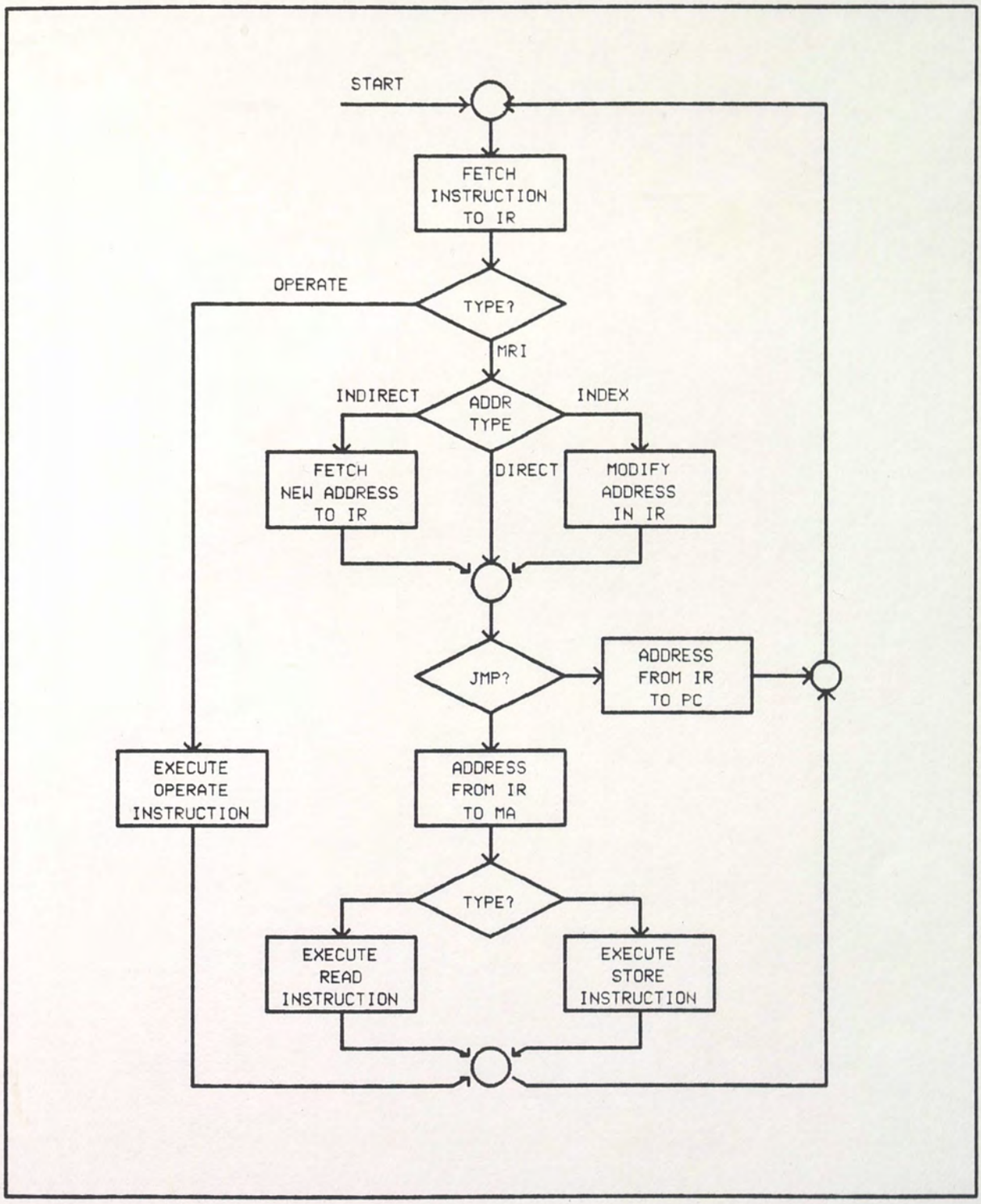

Figure 6. Class A Sic Instruction Flow.

```
get_word          := (MD = M[MA])
store_word        := (M[MA] = MD)

fetch_instruction := (MA = PC; next;
                      get_word;
                      IR = MD; PC = PC + 1; next)
```

The instruction model is developed in metaMicro from the instruction definitions given above. Since SIC has a single fixed length word instruction this is fairly straightforward. A macro is created to create the bit pattern of each instruction. Where there is commonality among instructions, sub-macros are referenced. For example, a macro to generate the address for the address field of the MRI instructions should be developed. The LAC (Load ACcumulator) instruction is presented as an example. A complete listing of the metaMicro instruction generation code is given in Appendix C.

```
lac(a,m)  = opcode = 1; addr = a; mode(m) $ &,
mode(m)   =
            if 'm eql "d" then {addr_type = direct};
            if 'm eql "i" then {addr_type = indirect};
            if 'm eql "a" then {addr_type = index_a};
            if 'm eql "b" then {addr_type = index_b}; &;
```

The class A implementation of the SIC machine reveals several interesting facts. Instruction set modeling using the N.2 software is very easy. The Pascal-like nature of ISP' makes performing functions a very easy task and instruction execution is nothing but the two-step process of deciding what function to perform and then performing it.

While the detail of this model may not seem great, much can be gained from it. The ISP' code makes an excellent instruction level description of the SIC machine that can be understood by both a system and hardware designer. With the inclusion of some minimal timing information, it can be used as a system specification. Because of the procedural nature of the code, modifications are easily performed. This allows a great amount of experimenting before any hardware is designed. With the inclusion of more detailed timing a good estimation of the performance of required algorithms can be made. In addition, the resultant code is far easier to read than non-procedural code like AHPL. Finally, after more detailed models are constructed, information can be fed back into this model. This allows the model to be used as a software emulator of the hardware in a software design system.

Class B Implementation of SIC

The class B model extends the facilities implemented in the class A model to include I/O and interrupt capabilities. The overall system is shown in Figure 7 and the more detailed SIC machine is shown in Figure 8. Since the purpose of the Class B model is to provide a pin level description, several models of external processes such as a main memory module and an I/O process, are developed.

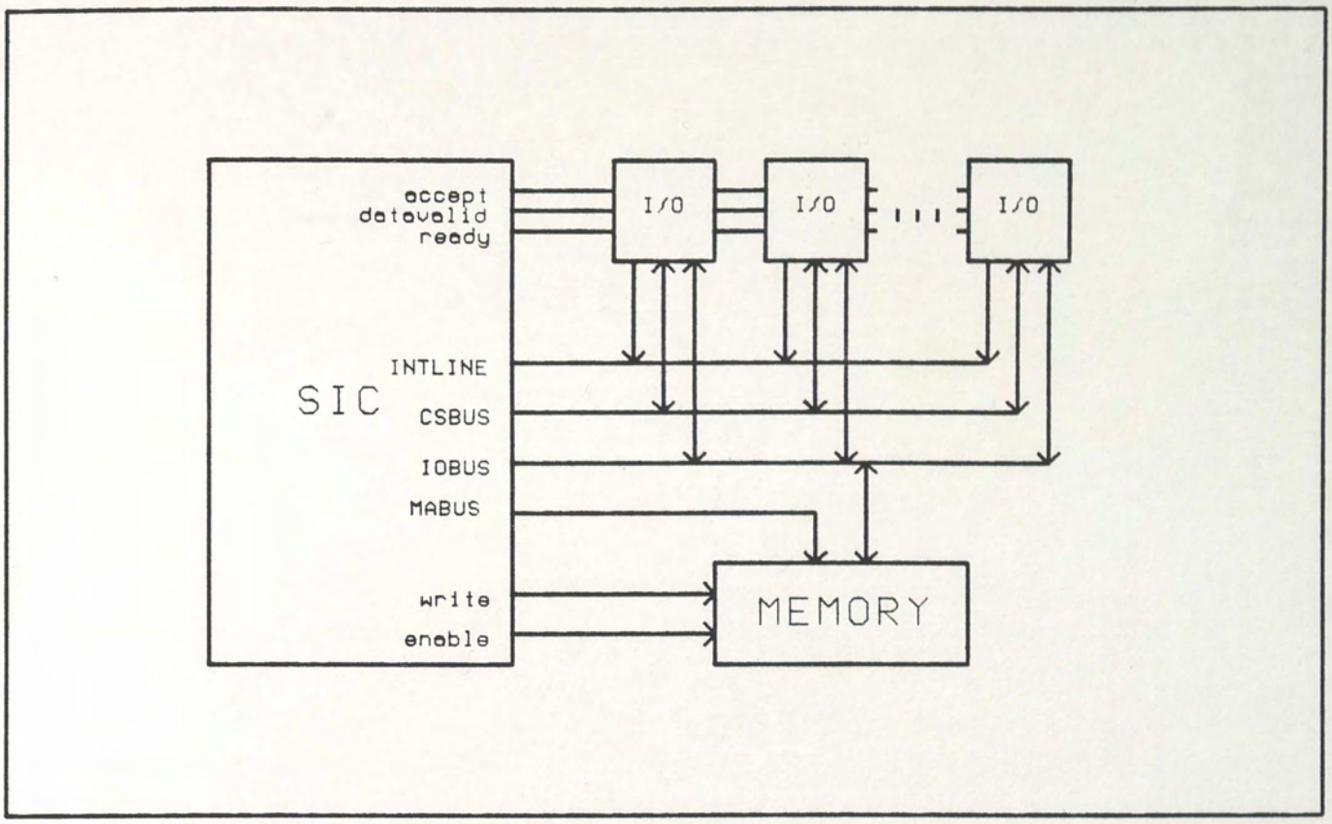

Figure 7. Class B Overall System.

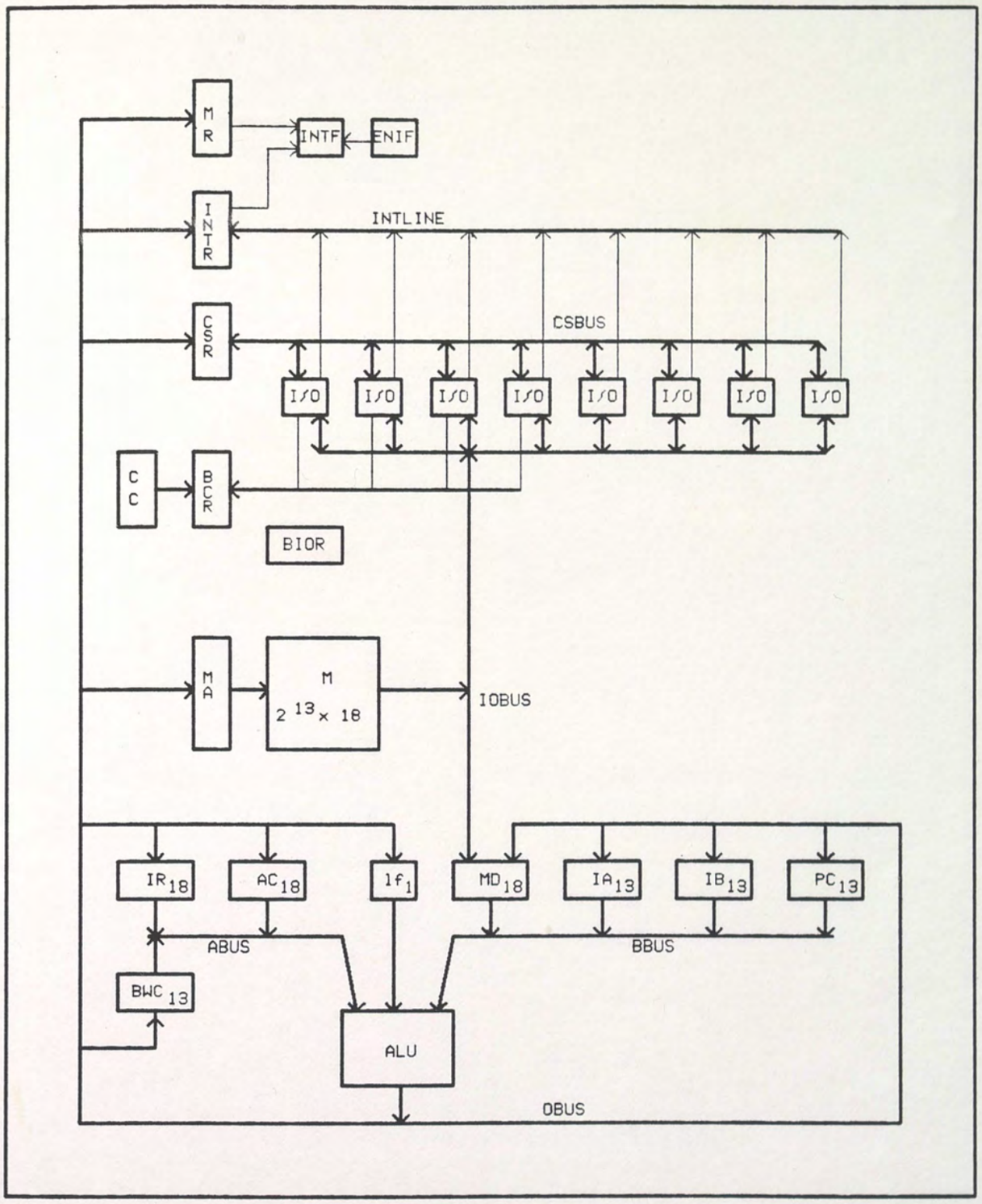

Figure 8. Class B SIC Facilities.

For the Class B model the instruction set is enhanced as is the handling of the OPERATE instructions. These are processed in three event times, instead of one in the Class A model. The format of the new instruction set is included in Figure 9, and the new instructions are defined in Table II. The event times for the OPERATE instructions are given in Table III. A flowchart for the handling of the new instructions is shown in Figure 10.

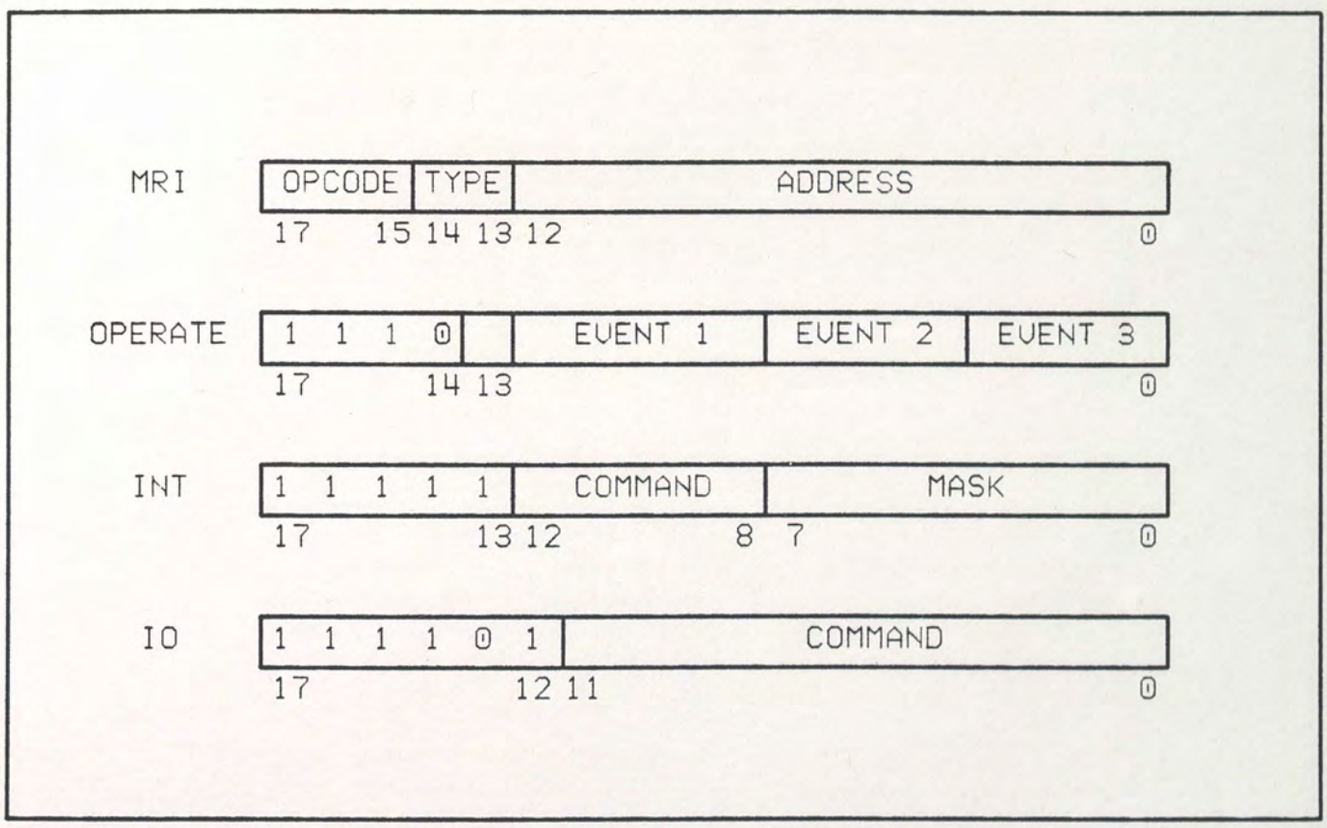

Figure 9. Class B SIC Instruction Format.

TABLE II

ADDITIONAL SIC INSTRUCTIONS

OPCODE	MEANING
LMI	load mask register from IR
LMA	load mask register from accumulator
LAM	load accumulator from mask register
MII	mask interrupt from IR
CLI	clear interrupt
EAI	enable interrupts
DAI	disable interrupts
ODn	output data to device n
IDn	input data from device n
ISn	input status from device n
OBn	activate output buffer to device n
IBn	activate input buffer to device n
OCn	output command to device n

TABLE III

EVENT TIME STRUCTURE OF OPERATE INSTRUCTIONS

EVENT	BIT FIELD	VALUE	MEANING
0	13	0	rotate direction is left
		1	rotate direction is right
1	12	0	no rotate
		1	rotate AC
	11:10	00	no op
		01	set link
		10	clear link
		11	halt
	9:8	00	no op
		01	set AC
		10	clear AC
		11	complement AC
2	7	0	no rotate
		1	rotate AC
	6:4	000	no op
		001	SZL, skip event 3
		010	DFA
		011	DFB
		100	DTA
		101	INA
		110	DTB
		111	INB
3	3	0	no rotate
		1	rotate AC
	2	0	no op
		1	skip if AC < 0
	1	0	no op
		1	skip if AC = 0
	0	0	no op
		1	skip if AC > 0

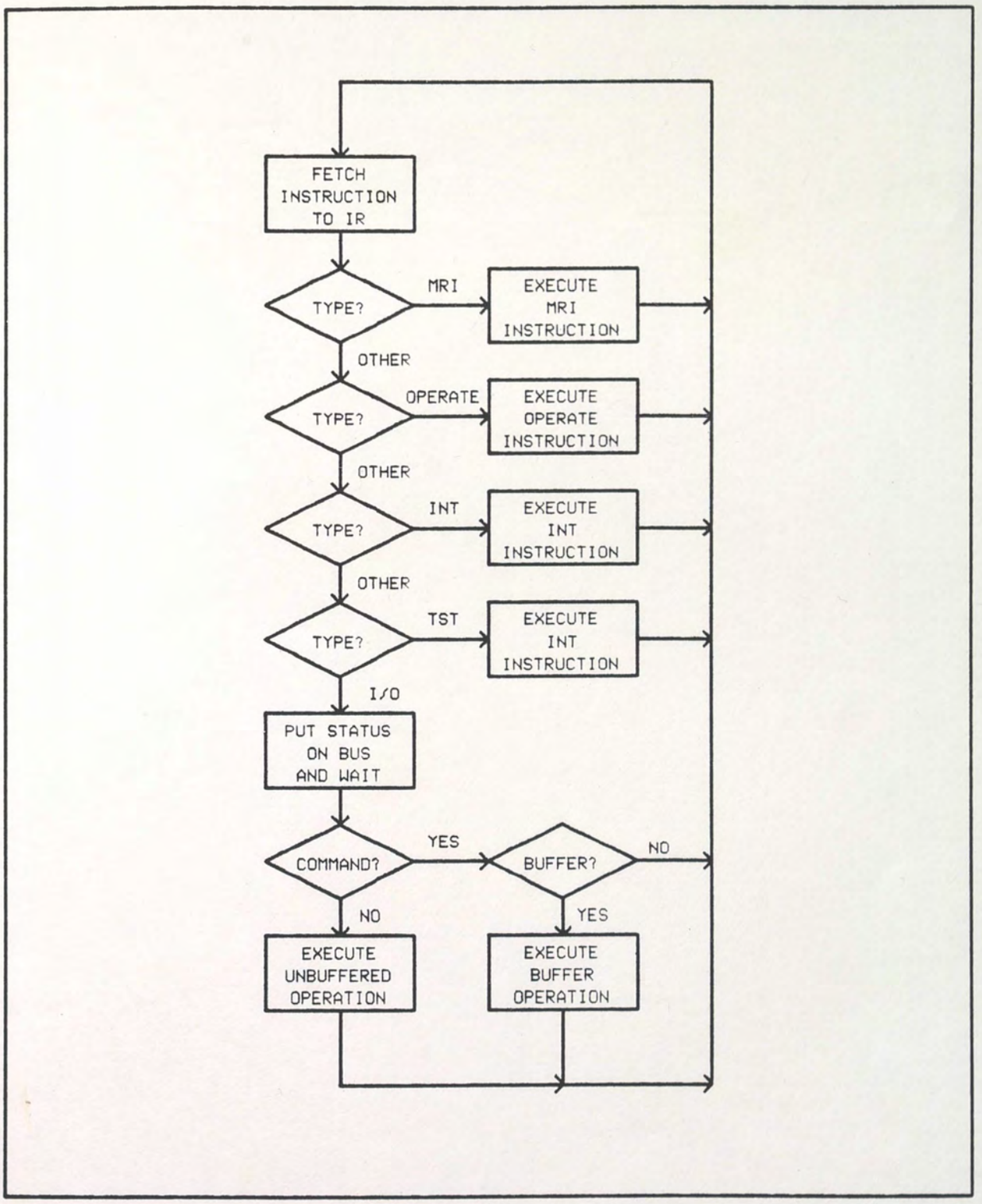

Figure 10. Class B SIC Flowchart.

In the Class B model it is assumed that each register transfer takes exactly one clock time to be performed. Thus each register operation in the ISP' source code has a CYCLE (delay(1)) command following it. The multiple event times used to implement the OPERATE instructions bring out these register operations more clearly. The complete ISP' source code is given in Appendix D.

As mentioned, models of processes external to SIC also need to be created. A synchronous memory module is presented to implement the main program memory. This module communicates to the SIC machine over four lines, illustrated in the port declarations of that module.

```
port    ADDR <ADDRSIZE>,        ! address bus
        DATA <DATASIZE>,        ! data bus
        write,                  ! read/write line
                                ! read = 0, write = 1
        enable;                 ! enable
```

The ADDR port connects to the MA register to the MABUS and contains the address of the word to be referenced in memory. The DATA port connects to the IOBUS of the SIC machine which feeds the memory data register and contains the data written/read to/from memory. An ENABLE line is used to enable memory, and a WRITE line is used to write to memory.

The I/O device developed emulates a hardware multiplier. It revceives data over the IOBUS from the SIC machine and stores it in

an internal register. Signalling of transfers is accomplished by using the lines READY, DATAVALID, and ACCEPT. The purpose of this process is to multiply that last two numbers received, faster than the SIC machine could perform the operation, but still in more than one clock time. The complete description of both the memory and I/O modules is given in Appendix E.

In order to connect these modules to the SIC machine the Ecologist is used. A topology file, given in Appendix F, is used to declare each of the modules and their port connections.

Since the instruction set was slightly enhanced, a more complete metaMicro description is required. This is given in Appendix G. Note the addition of several special commands, which are not part of the SIC assembly set, but which do test the use of multiple event times in the OPERATE instructions. The Class B model of SIC demonstrates N.2's capabilities at modeling near the register level. The ease with which the Class A model was enhanced to the Class B implementation shows how simpler models can be enhanced to more complicated ones without the need of a separate design language.

Class C Implementation of SIC

So far no mention of the control system for the SIC machine has been made. The C model is designed to address that issue. A micro-controller for the SIC machine was developed. The system is shown in

Figure 11. The microword format is shown in Figure 12.

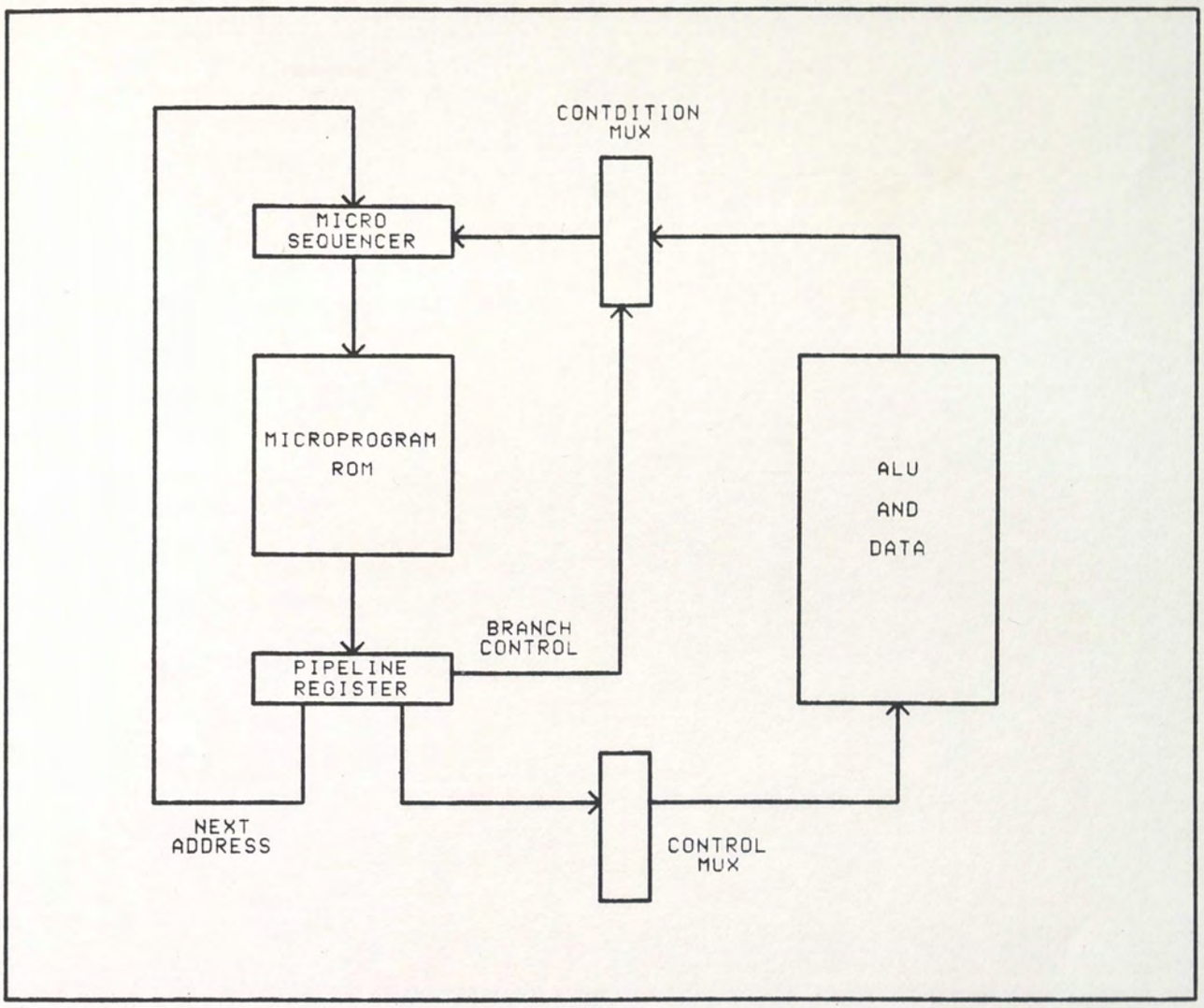

Figure 11. Class C SIC System.

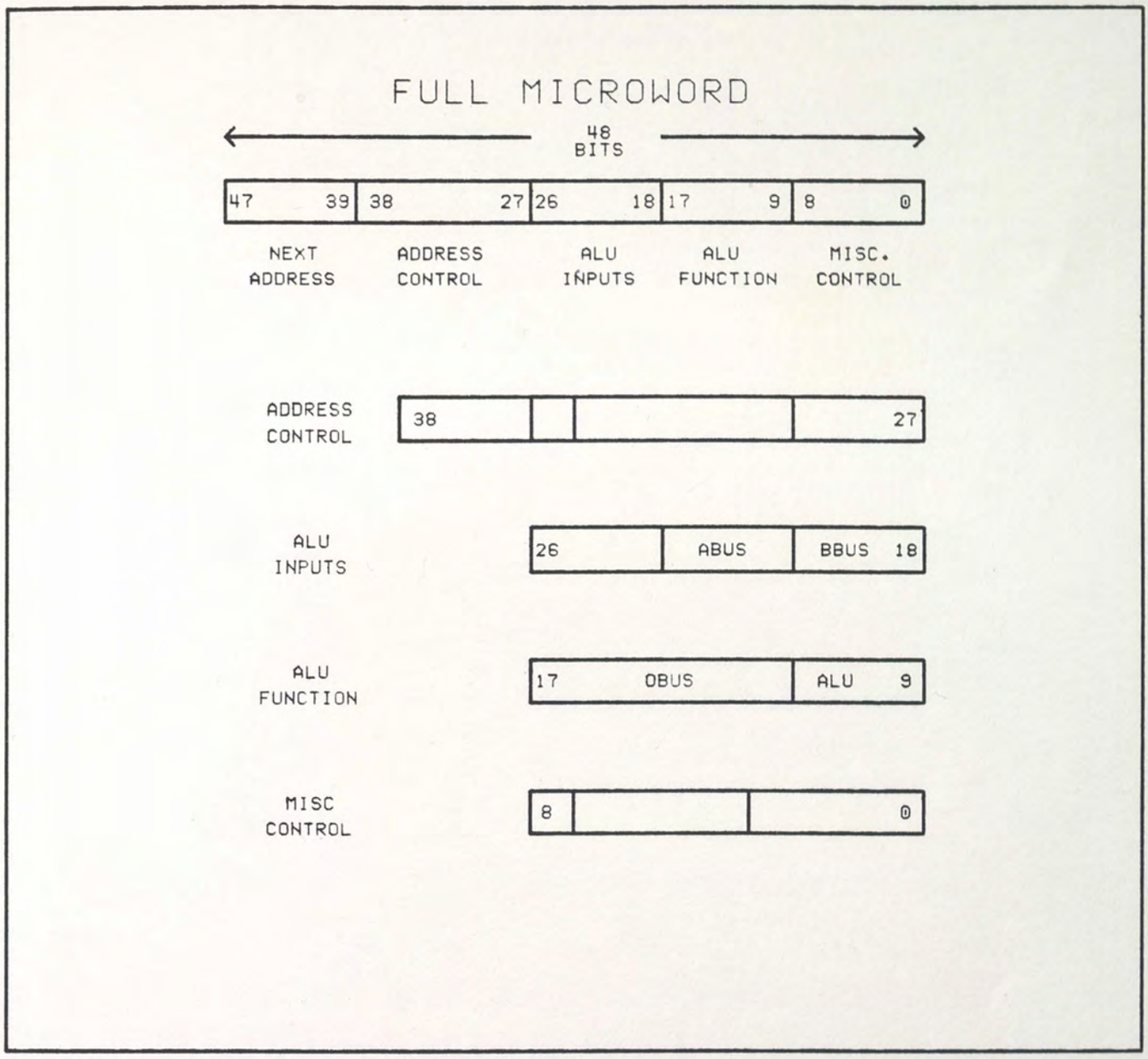

Figure 12. Class C SIC Microword Format.

Basically, a micro program ROM stores the microprogram word. On the leading edge of the clock this word is latched into the pipeline register. Two simultaneous processes then begin. The micro sequencer generates the next ROM address based on the next address field of the

current microword, and on the condition code generated from the SIC machine after it executed the last micro instruction. This occurs while the SIC machine is executing the present instruction. All registers in the SIC machine are latched on the falling edge of the clock, as is the address generated by the micro sequencer. This process is illustrated in the timing diagram shown in Figure 13.

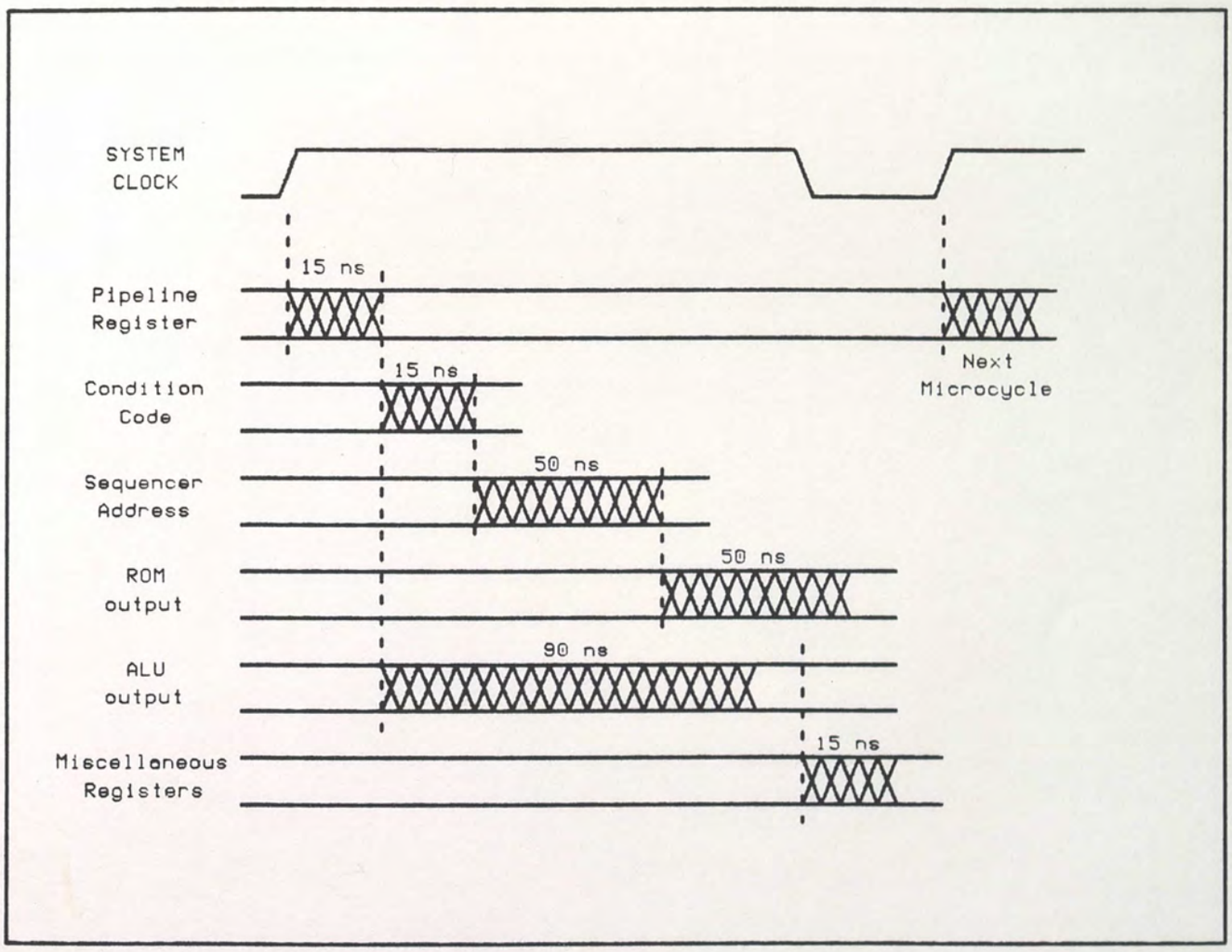

Figure 13. Class C SIC Microcycle Timing.

In order to specify the functions over which the micromachine has control, a much more detailed model of SIC is created. This is given in Appendix H. Each element of the SIC machine is modeled as an independent process. That is, each register, and its associated input and output multiplexers, are developed in separate ISP' files. Where more than one register is the same, like the IA, IB, and PC registers, multiple instances of a single model can be used.

The timing of the Class C machine consists of assigning a delay to each of the subprocesses that make up the machine. These delay numbers are specified in nanosecond units. The synchronization of the processes is accomplished with a clock generator model.

Due to the large number of communicating processes, the importance of the topology file increases. The topology file is given in Appendix I. The ability to connect ports to only part of a signal is used freely.

Two memories exist in the Class C SIC machine, the program RAM and the microprogram ROM. The metaMicro code used to set up the program RAM is the same as that used in the B level model. A separate description must be developed for the program ROM. Again, the instruction length of the micromachine is a constant one word, so program development is fairly straightforward. The Microprogram is not presented here.

The Class C SIC model was very difficult to construct, and probably illustrates the lowest level of modeling that can effectively be performed with N.2. If a library of ISP' primitives was present, for example a collection of registers, memories, muxes, etc., then modeling at this level would become far easier. As it is, however, the time required and the potential mistakes induced by creating each of the individul registers models makes the net gain from this degree of modeling questionable.

Not far from this level of modeling, is modeling with logic level simulators. Since these simulators typically contain registers, memories, and other primitives of this type, it seems that they should be used to perform this detail of simulation. A further advantage of logic level simulators is that coupled with a graphics entry front end, they provide releasable schematics in addition to simulation capabilities for hardware design.

An advantage that the ISP' modeling language has over logic simulators is that each of the ISP' primitives may be of any complexity. For example, a complicated processor function, such as multiplication, is easily coded in a single ISP' step. To implement complicated functional blocks with gates and registers wastes design time, if the intent is only to model that functional block for use in a larger simulation. In the final analysis, however, it seems that effective Class C models can only be developed by including the ability to

reference ISP' code by a logic simulator, or by developing a large logic level parts library for the ISP' simulator.

SECTION III

COMMENTS AND CONCLUSIONS

The N.2 system clearly provides a useful tool in digital design cycle. The major strong points of the software system are summarized below, along with a list of its shortcomings.

The modeling language ISP' provides excellent system modeling capabilities at the instruction or register level. A design can be partitioned into functional blocks and these blocks combined into a simulation. Delays can be included in the block specification to increase the accuracy of the simulation. The resultant ISP' code of the machine also provides a specification that a system level designer as well as a logic designer can understand and use.

The instruction specification language, metaMicro, allows customized assemblers to be developed. This greatly speeds the development of sample code and microcode for the target machine. The author was unable to locate another functional modeling tool with this capability.

Several enhancements to the capabilities of the N.2 system would greatly improve its capabilities. First, to increase the capabilities at the more abstract levels of simulation, signals should be data structures.

That is, the current 1 or 0 system that ISP' uses to reference signal values should be replaced with user-definable signal levels. This capability is already present in other functional level simulators such as HHDL*.

Second, the instrinsic functions able to be referenced in the ISP' code should be increased to include a greater subset of the PASCAL or FORTRAN function set. The inclusion of trigonometric fuctions along with data structured ports would make the design of communication systems possible using ISP'.

Third, to increase to power of the N.2 system at the gate level, a parts library needs to be developed. Parts should include registers, multiplexers, and various combinational logic primitives. In addition to generic primitives, a set of standard TTL parts should also be developed.

Finally, to increase the capabilities of the Ecologist as a binding tool for ISP' modules, parameters should be able to be passed to those modules. That is, parameters other than delay time and signal connections. This would allow for configurable parts such a N bit registers or M input gates.

---

*HHDL is the hardware design language used with the HELIX simulator used at Martin Marietta. This software is produced by Silvar Lisco Inc.

In general, the N.2 system provides a useful tool for the development of new processor systems early in the design cycle. It is limited in its capabilities later in the design cycle and would have to be replaced by another design tool.

APPENDIX A

SIC AHPL

The SIC control sequence given here was used to develop the ISP' models of the SIC machine. This includes the interrupt, I/O and buffer sequences, but does not include the DMA. Also the INT and TST sequences are also not given. A fully synchronous memory of one clock period has been assumed throughout.

```
MODULE : SIC
MEMORY  : M[8192;18]; AC[18]; MD[18]; IR[18]; PC[13]; MA[13];
          IA[13]; IB[13]; MR[8]; INTR[8]; CSR[12]; BWC[13];
          BCR[4]; BIOR[4]; CC[2]; If; intf; enif
INPUTS  : INTLINE[8]; BCRDY[4]; start; ready; dataval id; accept
OUTPUTS : BUFRDY[4]; csrdy; ready; dataval id; accept; bufend
BUSES   : ABUS[18]; BBUS[18]; OBUS[18]
COMBUSES: IOBUS[18]; CSBUS[12]

1.  ->(start)/(1)
2.  ->(v/BCR)/(90)
2.1 ->(intf)/(60)
2.2 MA <- PC; PC <- INC(PC)
3.  MD <- BUSFN(M; DCD(MA))
4.  IR <- MD; intf*((v/(MR ^ INTR)) ^ enif) <- 1
5.  ->(IR0 ^ IR1 ^ IR2)/(25)
6.  NO DELAY; ->((IR3 ^ IR4),(IR3 ^ IR4),IR3)/(13,7,10)
7.  MA <- IR5:17
8.  MD <- BUSFN(M;DCD(MA))
9.  IR5:17 <- MD5:17; -> (13)
10. NO DELAY; ->(IR4)/(12)
11. IR5:17  <- ADD(IR5:17;IA); ->(13)
```

```
12. IR5:17  <- ADD(IR5:17;IB)
13. NO DELAY; ->(IR0^ IR1)/(15)
14. PC <- IR5:17; ->(2)
15. MA <- IR5:17; ->(IR0)/(21)
16. MD <- BUSFN(M;DCD(MA)); ->(IR1 ^ IR2)/(18)
17. AC <- (MD ! (MD ^ AC ! ADD(MD;AC)))*
          ((IR1 ^ IR2),(IR1 ^ IR2),(IR1 ^ IR2));
    If*(IR1 ^ IR2) <- ADD0(MD;AC); ->(2)
18. MD <- INC(MD)
19. M*DCD(MA) <- MD; ->(V/MD)/(24)
20. PC <- INC(PC); ->(2)
21. MD <- (AC ! (5 T 0,INC(PC)))*(IR2,IR2)
22. M*DCD(MA) <- MD; ->(IR2)/(2)
23. PC <- IR
24. PC <- INC(PC); ->(2)
25. NO DELAY; ->(IR3)/(50)
26. NO DELAY; ->(IR5)/(30)
27. NO DELAY; ->((IR6 ^ IR7),(IR6 ^ IR7)/(1,29)
28. NO DELAY
29. AC <- ((18 T 0) ! (18 T 0) ! AC)*
    ((IR8 ^ IR9),(IR8 ^ IR9),(IR8 ^ IR9)
    If*(IR6) <- 0; If*(IR7) <- 1; ->(33)
30. ->(IR4)/(32)
31. If,AC <- AC,If; ->(33)
32. If,AC <- AC17,If,AC0:16
33. NO DELAY; -> (IR10)/(40)
34. NO DELAY; -> (DCD(IR11,IR12)/(35,37,38,39)
35. NO DELAY; -> (IR13 ^ If)/(43)
36. PC <- INC(PC); ->(2)
37. AC <- (5 T 0,IA ! 5 T 0,IB)*(IR13,IR13); ->(43)
38. IA <- ((AC)!(INC(IA)))*(IR13,IR); ->(43)
39. IB <- ((AC)!(INC(IB)))*(IR13,IR); ->(43)
40. NO DELAY; -> (IR4)/(42)
41. If,AC <- AC,If; ->(43)
42. If,AC <- AC17,If,AC0:16
43. NO DELAY; ->(IR14)/(45)
44. ->(f,f)/(36,2);
    {f = (AC<0 ^ IR15) V (AC=0 ^ IR16) V (AC>0 ^ IR17)}
45. NO DELAY; ->(IR4)/(47)
46. If,AC <- AC,If; ->(2)
47. If,AC <- AC17,If,AC0:16
50. -> (IR4)/(INT SEQ)
51. -> (IR5,IR5)/(70,TST SEQ)
60. intf,enif <- 0,0
```

```
61. IR5:17 <- ADDR(PRI(INTR^MR))
62. MA <- IR5:17; MD <- 5 T 0,PC
63. M * (DCD(MA) <- MD
64. PC <- IR5:17; -> (24)
70. CSR <- IR6:17
71. CSBUS = CSR; csrdy = 1; ->(accept)/(71)
72. NO DELAY; -> (IR9)/(74)
73. NO DELAY; -> (IR10,IR10)(24,85)
74. NO DELAY; -> (IR11)/(78)
75. MD <- AC
76. -> (ready)/(76)
77. IOBUS = MD; datavalid = 1; ->(accept,accept)/(24,77)
78. ready = 1; -> (datavalid)/(78)
79. CSR*IR10 <- CSBUS; MD*IR10 <- IOBUS
80. accept = 1; -> (datavalid)/(80)
81. NO DELAY; -> (IR10)/(83)
82. AC <- MD; ->(24)
83. NO DELAY; ->(V/IR12:17^CSR6:11))/(2)
84. PC <- INC(PC); ->(2)
85. BIOR * DCD(IR7:8) <- (IR11 ^ 4 T 0); ->(24)
90. ->(V(DCD(CC)^BCR)/(92)
91. CC <- INC(CC); ->(90)
92. IR <- BADDR(CC); BCR*DCD(CC) <- 4 T 0
93. MA <- IR5:17
94. MD <- BUSFN(M;DCD(MA)); IR5:17 <- INC(IR5:17)
95. MA <- IR5:17; BWC <- MD
96. MD <- BUSFN(M;DCD(MA))
97. MA <- ADD(MD;BWC)
98. BWC <- INC(BWC); BUFRDY = DCD(CC)
99. NO DELAY; ->(V/(DCD(CC)^BIOR)/(103)
100. MD <- BUSFN(M;DCD(MA))
101. ->(ready)/(101)
102. IOBUS = MD; datavalid = 1; ->(accept,accept)/(107,102)
103. ready = 1; ->(datavalid)/(103)
104. MD <- IOBUS
105. M * DCD(MA) <- MD
106. accept = 1; ->(datavalid)/(106)
107. NO DELAY; ->(V/BWC)/(109)
108. bufend = 1; ->(2)
109. BUFRDY = DCD(CC)
110. MA <- BADDR(CC); MD <- BWC;
111. M*DCD(MA) <- MD
```

APPENDIX B

"A" MODEL ISP' SOURCE FOR SIC

The following ISP' source code represents the "A" level model for SIC. A very limited subset of the SIC features are included. These are the MRI and OPERATE instructions. No timing information, or port information is contained in this model.

```
! ***************************************************************
!
!  Name     : ASIC.ISP
!  Purpose  : ISP' code for a
!             Small Instruction set Computer,
!             class A implementation
!  Author   : BJ Patz
!  Version  : 1.0
!
!  Comments : MRI and OPERATE instructions modeled only
!             no ports are modeled
!             program memory included
!
! ***************************************************************

!
!  declarations
!

macro    WORD      = 18 &,          ! basic word size
         ADDRESS   = 13 &,          ! basic address length
         ADDR_PART = 12:0 &,        ! address part of WORD
         CYCLE     = delay(1) &,    ! basic cycle time
                                    ! one cycle per instruction
```

```
! major instruction breakdown, bits 17:15

        ISZ_I     = 0 &,          ! inc and skip on zero
        LAC_I     = 1 &,          ! load AC
        AND_I     = 2 &,          ! and MD with AC
        TAD_I     = 3 &,          ! twos comp add MD with AC
        JMS_I     = 4 &,          ! jump to subroutine
        DAC_I     = 5 &,          ! deposit AC
        JMP_I     = 6 &,          ! jump
        OP_IO_I   = 7 &,          ! operate or I/O instruction

! operate instruction breakdown, bits 12:0

        HLT_I     = 0x0c00 &,     ! halt
        NOP_I     = 0x0000 &,     ! no operation
        CLA_I     = 0x0200 &,     ! clear accumulator
        STA_I     = 0x0100 &,     ! set accumulator
        CMA_I     = 0x0300 &,     ! complement accumulator
        CLL_I     = 0x0800 &,     ! clear link
        STL_I     = 0x0400 &,     ! set link
        SKP_I     = 0x0003 &,     ! skip if accum >= 0
        SKZ_I     = 0x0002 &,     ! skip if accum = 0
        SZL_I     = 0x0010 &,     ! skip if link = 0
        RAR_I     = 0x3000 &,     ! rotate accum right
        RAL_I     = 0x2000 &,     ! rotate accum left
        DTA_I     = 0x0040 &,     ! deposit accum to IA
        DTB_I     = 0x0060 &,     ! deposit accum to IB
        DFA_I     = 0x0020 &,     ! deposit IA to accum
        DFB_I     = 0x0030 &,     ! deposit IB to accum
        INA_I     = 0x0050 &,     ! increment IA
        INB_I     = 0x0070 &,     ! increment IB

! adressing modes

        DIRECT    = 0 &,          ! direct addressing
        INDIRECT  = 1 &,          ! indirect addressing
        INDEX_A   = 2 &,          ! index A addressing
        INDEX_B   = 3 &,          ! index B addressing

        xw(val)  = val ext 18 &;

state   AC   <WORD>,              ! accumulator
        MD   <WORD>,              ! memory data register
        IR   <WORD>,              ! instruction register
```

```
        PC   <ADDRESS>,              ! program counter
        MA   <ADDRESS>,              ! memory address register
        IA   <ADDRESS>,              ! index register A
        IB   <ADDRESS>,              ! index register B
        lf;                          ! link flag

memory  M [0:8191] <WORD>;           ! program memory

format  opcode      = IR<17:15>,     ! operation
        addr_type   = IR<14:13>,     ! address type
        addr        = IR<12:0>,      ! address
        op_part     = IR<13:0>;      ! operate part of instruction

!
! sub processes
!

! ********************************************************************
! Memory operations

get_word          := (MD = M[MA])
store_word        := (M[MA] = MD)

fetch_instruction := (MA = PC; next;
                      get_word;
                      IR = MD; PC = PC + 1; next)
load_md           := (MA = addr; next;
                      get_word)

! ********************************************************************
! compute effective address

effective_address :=
        (
         MA = addr; next;
         case addr_type
              direct   : ;
              indirect : (get_word; addr = MD<ADDR_PART>)
              index_a  : (addr = addr + IA)
              index_b  : (addr = addr + IB)
         esac;
        )

! ********************************************************************
! perform MRI instructions
```

```
do_isz := (load_md;                          ! load MD
           MD = MD + 1; next;                ! increment MD
           store_word;                       ! store MD
           if (MD eql 0) (PC = PC+1))        ! if MD = 0 then skip
do_lac := (load_md;                          ! load MD
           AC = MD)                          ! load AC from MD
do_and := (load_md;                          ! load MD
           AC = MD and AC)                   ! AND AC and MD
do_tad := (load_md;                          ! load MD
           AC = MD + AC)                     ! add AC and MD
do_jms := (MD = xw(PC); next;
           store_word;                       ! save PC
           PC = addr)                        ! jump to address
do_dac := (MD = AC; next;                    ! load MD from AC
           store_word)                       ! save AC
do_jmp := (PC = addr)                        ! jump

! ************************************************************************
! perform OPERATE instructions

do_cla := (AC = 0)                           ! clear AC
do_sta := (AC = not 0)                       ! set AC
do_cma := (AC = not AC)                      ! complement AC
do_cll := (lf = 0)                           ! clear link
do_stl := (lf = 1)                           ! set link
do_skp := (if (AC geq 0) (PC = PC+1))        ! skip next instruction
do_skz := (if (AC eql 0) (PC = PC+1))        ! skip next instruction
do_szl := (if (lf eql 0) (PC = PC+1))        ! skip next instruction
do_rar := (AC = lf concat AC<(WORD-1_:0>;
           lf = AC<0>)                       ! rotate right
do_ral := (AC = AC concat lf;
           lf = AC<(WORD-1)>)                ! rotate left
do_dta := (IA = AC<addr_part>)               ! deposit AC in IA
do_dtb := (IB = AC<addr_part>)               ! deposit AC in IB
do_dfa := (AC = xw(IA))                      ! deposit IA in AC
do_dfb := (AC = xw(IB))                      ! deposit IB in AC
do_ina := (IA = IA + 1)                      ! increment IA
do_inb := (IB = IB + 1)                      ! increment IB

! ************************************************************************
! main program
!

main :=
```

```
(
 fetch_instruction;

 if (opcode neq OP_IO_I) effective_address;

 case opcode
      ISZ_I : (do_isz)
      LAC_I : (do_lac)
      AND_I : (do_and)
      TAD_I : (do_tad)
      JMS_I : (do_jms)
      DAC_I : (do_dac)
      JMP_I : (do_jmp)
      OP_IO_I :
               (
                case op_part
                     HLT_I : (;)
                     NOP_I : (;)
                     CLA_I : (do_cla)
                     STA_I : (do_sta)
                     CMA_I : (do_cma)
                     CLL_I : (do_cll)
                     STL_I : (do_stl)
                     SKP_I : (do_skp)
                     SKZ_I : (do_skz)
                     SZL_I : (do_szl)
                     RAR_I : (do_rar)
                     RAL_I : (do_ral)
                     DTA_I : (do_dta)
                     DTB_I : (do_dtb)
                     DFA_I : (do_dfa)
                     DFB_I : (do_dfb)
                     INA_I : (do_ina)
                     INB_I : (do_inb)
                esac;
               )

 esac;
 CYCLE;
)
```

APPENDIX C

"A" MODEL METAMICRO SOURCE FOR SIC

The following source code represents the metaMicro code for the "A" level model for SIC. Macros are included for every MRI and OPERATE instruction.

```
! ***************************************************************
!
!  Name      : ASIC.METAMICRO
!  Purpose   : metaMicro assembly code generator for SIC
!              class A implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments :
!
! ***************************************************************

instr    I[1,1]<18> $                ! 1 word instruction of 18 bits

format   opcode      = I<17:15>,     ! operation
         addr_type   = I<14:13>,     ! address type
         addr        = I<12:0>,      ! address

         rot_dir     = I<13:13>,     ! rotation direction
         rot1        = I<12:12>,     ! rotate in event 1
         rot2        = I<7:7>,       ! rotate in event 1
         rot3        = I<3:3>,       ! rotate in event 1
         ev1_1       = I<11:10>,     ! event time 1 sub event 1
         ev1_2       = I<9:8>,       ! event time 1 sub event 2
         ev2         = I<6:4>,       ! event time 2 only event
         ev3_lt      = I<2:2>,       ! event time 3 less than 0
         ev3_eq      = I<1:1>,       ! event time 3 equal 0
```

```
        ev3_gt      = I<0:0>,      ! event time 3 greater than 0

        op_io_int   = I<14:12>,    ! operate, I/O, and interrupt
                                   ! opcodes

        io_device   = I<11:9>,     ! I/O device
        io_command  = I<8:7>,      ! I/O command
        io_status   = I<11:0>,     ! I/O status
        io_data     = I<7:7>,      ! I/O data or status
        io_dir      = I<6:6>,      ! I/O direction
        io_comp     = I<5:0>,      ! I/O status compare

        buf_io_chan = I<10:9>$     ! buffer I/O channel

macro

! constants

        direct    = 0 &,
        indirect  = 1 &,
        index_a   = 2 &,
        index_b   = 3 &,
        left      = 0 &,
        right     = 1 &,

! the basic op codes

        isz(a,m)  = opcode = 0; addr = a; mode(m) $ &,
        lac(a,m)  = opcode = 1; addr = a; mode(m) $ &,
        and(a,m)  = opcode = 2; addr = a; mode(m) $ &,
        tad(a,m)  = opcode = 3; addr = a; mode(m) $ &,
        jms(a,m)  = opcode = 4; addr = a; mode(m) $ &,
        dac(a,m)  = opcode = 5; addr = a; mode(m) $ &,
        jmp(a,m)  = opcode = 6; addr = a; mode(m) $ &,

        ral       = opcode = 7; rot_dir = left; rot1 = 1;
        rar       = opcode = 7; rot_dir = right; rot1 = 1;

        nop       = opcode = 7; ev1_1 = 0 $ &,
        stl       = opcode = 7; ev1_1 = 1 $ &,
        cll       = opcode = 7; ev1_1 = 2 $ &,
        hlt       = opcode = 7; ev1_1 = 3 $ &,

        sta       = opcode = 7; ev1_2 = 1 $ &,
        cla       = opcode = 7; ev1_2 = 2 $ &,
```

```
        cma         = opcode = 7; ev1_2 = 3 $ &,

        szl         = opcode = 7; ev2 = 1 $ &,
        dfa         = opcode = 7; ev2 = 2 $ &,
        dfb         = opcode = 7; ev2 = 3 $ &,
        dta         = opcode = 7; ev2 = 4 $ &,
        ina         = opcode = 7; ev2 = 7 $ &,
        dtb         = opcode = 7; ev2 = 6 $ &,
        inb         = opcode = 7; ev2 = 7 $ &,

        skp         = opcode = 7; ev3_eq = 1; ev3_gt = 1 $ &,
        skz         = opcode = 7; ev3_eq = 1; $ &,

! address mode determination

        mode(m)     =
                        if 'm eql "d" then {addr_type = direct};
                        if 'm eql "i" then {addr_type = indirect};
                        if 'm eql "a" then {addr_type = index_a};
                        if 'm eql "b" then {addr_type = index_b}; &;
```

APPENDIX D

"B" MODEL ISP' SOURCE FOR SIC

The following ISP' source code represents the "B" level model for SIC. The complete instruction set of SIC, as defined by the AHPL code in Appendix A, is modeled. Interconnection capabilities to an external memory and I/O facilities are also provided.

```
! ***************************************************************
!
!  Name      : BSIC.ISP
!  Purpose   : ISP' code for a Small Instruction set Computer,
!              class B
!              implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments : all instruction groups modeled
!
! ***************************************************************

!
!  declarations
!

macro    WORD       = 18 &,          ! basic word size
         ADDRESS    = 13 &,          ! basic address length
         ADDR_PART  = 12:0 &,        ! address part of WORD
         STATUS     = 12 &,          ! status length
         BUF_CHAN   = 4 &,           ! number of buffer channels
         INT_CHAN   = 8 &,           ! number of interrupt lines
         STAT_PART  = 5:0 &,         ! comparable part of STATUS
         CYCLE      = delay(1) &,    ! basic cycle time
```

```
! major instruction breakdown, bits 17:15

        ISZ_I     = 0 &,          ! inc and skip on zero
        LAC_I     = 1 &,          ! load AC
        AND_I     = 2 &,          ! and MD with AC
        TAD_I     = 3 &,          ! twos comp add MD with AC
        JMS_I     = 4 &,          ! jump to subroutine
        DAC_I     = 5 &,          ! deposit AC
        JMP_I     = 6 &,          ! jump
        OP_IO_I   = 7 &,          ! operate or I/O instruction

! adressing modes

        DIRECT    = 0 &,          ! direct addressing
        INDIRECT  = 1 &,          ! indirect addressing
        INDEX_A   = 2 &,          ! index A addressing
        INDEX_B   = 3 &,          ! index B addressing

        LEFT      = 0 &,
        RIGHT     = 1 &,


        xw(val)  = val ext 18 &;

port    INTLINE <INT_CHAN>,       ! interrupt lines
        BCRDY   <BUF_CHAN>,       ! buffer channel ready?
        BUFRDY  <BUF_CHAN>,       ! buffer ready
        start,                    ! start signal
        csrdy,                    ! status ready
        ready,                    ! io ready
        datavalid,                ! io data valid
        accept,                   ! io accepted
        bufend,                   ! buffer done
        IOBUS  <WORD>,            ! io info bus, memory data bus
        CSBUS  <STATUS>,          ! status bus
        MABUS  <ADDRESS>,         ! memory address bus
        mwrite,                   ! memory write
        menable;                  ! memory enable

state   AC   <WORD>,              ! accumulator
        MD   <WORD>,              ! memory data register
        IR   <WORD>,              ! instruction register
        PC   <ADDRESS>,           ! program counter
        MA   <ADDRESS>,           ! memory address register
        IA   <ADDRESS>,           ! index register A
```

```
        IB   <ADDRESS>,             ! index register B
        lf,                         ! link flag
        MR   <INT_CHAN>,            ! mask register
        INTR <INT_CHAN>,            ! interrupt register
        CSR  <STATUS>,              ! comm status register
        BWC  <ADDRESS>,             ! buffer word count
        BCR  <BUF_CHAN>,            ! buffer channel register
        BIOR <BUF_CHAN>,            ! buffer I/O direction
        CC   <1:0>,                 ! 2 bit counter
        intf,                       ! interrupt flag
        enif;                       ! interrupt enable

format  opcode      = IR<17:15>,    ! operation
        addr_type   = IR<14:13>,    ! address type
        addr        = IR<12:0>,     ! address

        rot_dir     = IR<13:13>,    ! rotation direction
        rot1        = IR<12:12>,    ! rotate in event 1
        rot2        = IR<7:7>,      ! rotate in event 1
        rot3        = IR<3:3>,      ! rotate in event 1
        ev1_1       = IR<11:10>,    ! event time 1 sub event 1
        ev1_2       = IR<9:8>,      ! event time 1 sub event 2
        ev2         = IR<6:4>,      ! event time 2 only event
        ev3_lt      = IR<2:2>,      ! event time 3 less than 0
        ev3_eq      = IR<1:1>,      ! event time 3 equal 0
        ev3_gt      = IR<0:0>,      ! event time 3 greater than 0

        op_io_int   = IR<14:12>,    ! operate, I/O, and interrupt
opcodes

        io_device   = IR<11:9>,     ! I/O device
        io_command  = IR<8:7>,      ! I/O command
        io_status   = IR<11:0>,     ! I/O status
        io_data     = IR<7:7>,      ! I/O data or status
        io_dir      = IR<6:6>,      ! I/O direction
        io_comp     = IR<5:0>,      ! I/O status compare

        buf_io_chan = IR<10:9>;     ! buffer I/O channel


!
! sub processes
!
```

```
! ***************************************************************
! get a word from memory, 1 cycle read

get_word :=
        (
         menable = 1;mwrite = 0; MABUS = MA; MD = IOBUS; CYCLE;
         menable = 0;
        )

! ***************************************************************
! store a word, 1 cycle write

store_word :=
        (
         menable = 1;mwrite = 1; MABUS = MA; IOBUS = MD; CYCLE;
         menable = 0;
        )

! ***************************************************************
! fetch an instruction

fetch_instruction :=
        (
         MA = PC; CYCLE;
         get_word;
         IR = MD; PC = PC + 1; CYCLE;
        )

! ***************************************************************
! compute effective address

effective_address :=
        (
         MA = addr; CYCLE;
         case addr_type
              direct   : ;
              indirect : (get_word; addr = MD<ADDR_PART>;CYCLE)
              index_a  : (addr = addr + IA; CYCLE)
              index_b  : (addr = addr + IB; CYCLE)
         esac;
        )

! ***************************************************************
! load MD from address in IR
```

```
load_md :=
        (
         MA = addr; CYCLE;
         get_word;
        )

! ********************************************************************
! perform MRI instructions

do_isz := (load_md;                          ! load MD
           MD = MD + 1; next;                ! increment MD
           store_word;                       ! store MD
           if (MD eql 0) (PC = PC+1))        ! if MD = 0 then skip
do_lac := (load_md;                          ! load MD
           AC = MD)                          ! load AC from MD
do_and := (load_md;                          ! load MD
           AC = MD and AC)                   ! AND AC and MD
do_tad := (load_md;                          ! load MD
           AC = MD + AC)                     ! add AC and MD
do_jms := (MD = xw(PC); next;
           store_word;                       ! save PC
           PC = addr)                        ! jump to address
do_dac := (MD = AC; next;                    ! load MD from AC
           store_word)                       ! save AC
do_jmp := (PC = addr)                        ! jump

! ********************************************************************
! rotate accumulator

rotate_ac :=
        (
         case rot_dir
              left  : (AC = AC concat lf; lf = AC<(WORD-1)>;
                       CYCLE)
              right : (AC = lf concat AC<(WORD-1):0>; lf = AC<0>;
                       CYCLE)
         esac;
        )

! ********************************************************************
! event time 1 for operate instructions

event1 :=
        (
```

```
          case rot1
               0 : (
                    case ev1_1
                         0 : ;                    ! nop
                         1 : (If = 1; CYCLE) ! set link
                         2 : (If = 0; CYCLE) ! clear link
                         3 : ;                    ! halt
                    esac;
                    case ev1_2
                         0 : ;                         ! nop
                         1 : (AC = not 0; CYCLE)  ! set AC
                         2 : (AC = 0; CYCLE)      ! clear AC
                         3 : (AC = not AC; CYCLE) ! comp AC
                    esac;
                   )
               1 : rotate_ac
          esac;
         )

! ***************************************************************
! event time 2

event2 :=
         (
          case rot2
               0 : (
                    case ev2
                         0 : ;
                         1 : (if If eql 0 then (PC = PC + 1; CYCLE)
                         2 : (AC = xw(IA); CYCLE) ! DFA
                         3 : (AC = xw(IB); CYCLE) ! DFB
                         4 : (IA = AC<ADDR_PART>; CYCLE) ! DTA
                         5 : (IA = IA + 1; CYCLE) ! INA
                         6 : (IB = AC<ADDR_PART>; CYCLE) ! DTB
                         7 : (IB = IB + 1; CYCLE) ! INA
                    esac
                   )
               1 : rotate_ac
          esac;
         )

! ***************************************************************
! event time 3
```

```
event3 :=
        (
         case rot3
              0 : (
                   if (
                       (ev3_lt and (AC lss xw(0))) or ! AC < 0
                       (ev3_eq and (AC eql xw(0))) or ! AC = 0
                       (ev3_gt and (AC gtr xw(0)))    ! AC > 0
                      ) (PC = PC+1; CYCLE)
                  )
              1 : rotate_ac
         esac;
        )

! ***************************************************************
! interrupt setup sequence not yet implemented

int :=
        (
         CYCLE
        )

! ***************************************************************
! interrupt priority logic, pick the most significant bit set

function pri(a<INT_CHAN>)<2:0> :=
        (
         state i <2:0>;

         i = 7; next;
         while (a<i:i> eql 0) (i = i-1);
         pri = i;
        )

! ***************************************************************
! compute interrupt address

function iaddr<ADDRESS> :=
        (
         iaddr = 8 + 2*(pri(INTR and MR) ext ADDRESS);
        )

! ***************************************************************
! service interrupt
```

```
int_handle :=
        (
         intf = 0; enif = 0; CYCLE:          ! reset interrupt flag,
                                             ! disable other
                                             ! interrupts
         addr = iaddr;                       ! get interrupt address
         MA = addr; MD = xw(PC); CYCLE;      ! store PC for return
         store_word;                         !
         PC = addr + 1; CYCLE;               ! jump to interrupt
                                             ! address + 1
        )

! ***************************************************************
! test sequence not yet implemented

tst :=
        (
         CYCLE
        )

! ***************************************************************
! I/O instructions

io :=
        (
         CSR = io_status; CYCLE;     ! put status on bus
         CSBUS = CSR; csrdy = 1; wait (accept : lead);
         CSBUS = 0; csrdy = 0; next; ! release bus
         case io_command
              0,1 : (
                     case io_dir
                          0 : (
! get data and wait until ready
                               MD = AC; CYCLE;
                               wait (ready : lead);
! put data on bus and wait until accepted
                               IOBUS = MD; datavalid = 1;
                               wait (accept : lead);
! release bus
                               IOBUS = 0; datavalid = 0;
                              )
                         1 : (
! ready to receive
                               ready = 1; wait(datavalid : lead);
```

```
                                    case io_data

! receive data
                                         0 : (MD = IOBUS; CYCLE)

! receive status
                                         1 : (CSR = CSBUS; CYCLE)
                                    esac;
                                    accept = 1;
                                    wait(datavalid : trail);
                                    case io_data
                                         0 : (AC = MD; CYCLE)
                                         1 : (
                                               if ((io_comp and
                                               CSR<STAT_PART>) neq 0)
                                                  (PC = PC + 1; CYCLE)
                                              );
                                    esac;
                                    accept = 0; ready = 0;
                                   )
                        esac
                       )
               2    : (   ! set buffered I/O direction
                        BIOR<buf_io_chan:buf_io_chan> = io_dir; CYCLE
                       )
               3    : ;   ! nop
          esac
         )

! ***************************************************************
! compute buffer address

function baddr(c<1:0>)<ADDRESS> :=
         (
          baddr = 32 + 2*c;
         )

! ***************************************************************
! buffer sequence

buffer :=
         (
          while BCR<CC:CC> eql 0    ! find out which channel
                (                   ! wants service
                 CC = CC + 1; CYCLE
                );
```

```
        addr = baddr(CC); BCR<CC:CC> = 0; CYCLE; ! get buffer
                                                 ! address
        MA = addr; CYCLE;
        get_word; addr := addr+1; next;          ! get negative
                                                 ! word count
        MA = addr; BWC = MD<ADDR_PART>; CYCLE;
        get_word;                                ! get start
                                                 ! address
        MA = MD + BWC; CYCLE;
        BWC = BWC + 1; BUFRDY<CC:CC> = 1; CYCLE; ! increment
                                                 ! word count
        BUFRDY<CC:CC> = 0;
        case BIOR<CC:CC>
             0 : (
                  get_word;                      ! send data
                  wait (ready:lead);
                  IOBUS = MD; datavalid = 1; wait(accept:lead);
                  IOBUS = 0; datavalid = 0;      ! release bus
                 )
             1 : (
                  ready = 1; wait(datavalid:lead);! receive data
                  MD = IOBUS; CYCLE;
                  store_word;
                  accept = 1; wait (datavalid:trail);
                  ready = 0; accept = 0;         ! release bus
                 )
        esac;
        if (BWC eql 0)
             (
              bufend = 1; CYCLE                  ! buffer empty
              bufend = 0;
             )
        else (
              BUFRDY<CC:CC> = 1; CYCLE;          ! store new BWC
              BUFRDY<CC:CC> = 0;
              MA = baddr(cc); MD = BWC; CYCLE;
              store_word;
             )
    )

! ********************************************************************
! main program
!
```

```
main :=
        (
         while BCR buffer;       ! buffer command
         if intf int_handle;     ! handle interrupt

         fetch_instruction;

         if (opcode neq OP_IO_I) effective_address;

         case opcode
              ISZ_I : (do_isz)
              LAC_I : (do_lac)
              AND_I : (do_and)
              TAD_I : (do_tad)
              JMS_I : (do_jms)
              DAC_I : (do_dac)
              JMP_I : (do_jmp)
              OP_IO_I :
                      (
                       case op_io_int

! operate instructions

                                0,1,2,3 :
                                    (
                                     event1;
                                     event2;
                                     if (ev2 neq 1) event3;
                                    )
! test sequence

                                4 : (
                                     tst;
                                    )

! io sequence

                                5 : (
                                     io;
                                    )
! interrupt sequence

                                6,7 :
                                    (
                                     int;
```

```
                    )
                esac;
            )

        esac;
        CYCLE;
    )
```

APPENDIX E

"B" MODEL ISP' SOURCE FOR MEMORY AND IO

The following ISP' source code represents a synchronous memory and an I/O module for the "B" level model of SIC.

```
! ***************************************************************
!
!  Name      : RAM.ISP
!  Purpose   : ISP' code for an RAM, used in the
!              Small Instruction set Computer
!              class B and C implementations
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments : RAM has enable to enable input and output
!
! ***************************************************************

!
!  declarations
!

macro     DATASIZE  = 18 &,          ! basic word size
          ADDRSIZE  = 13 &,          ! basic address length
          RAMDELAY  = 50 &;          ! RAM read delay

memory    M[0:8191]<DATASIZE>;       ! program memory

port      ADDR <ADDRSIZE>,           ! address bus
          DATA <DATASIZE>,           ! data bus
          write,                     ! read/write line
                                     ! read = 0, write = 1
          enable;                    ! enable
```

```
! ***************************************************************
! memory read, and write

do_read :=
        (
         delay(RAMDELAY);
         DATA = M[ADDR]
        )
do_write :=
        (
         M[ADDR] = DATA
        )

! ***************************************************************
! main routines

when (write  : trail (enable eql 1)) := (do_read)
when (write  : lead  (enable eql 1)) := (do_write)
when (enable : trail                 := (DATA = 0)
when (enable : lead  (write eql 0))) := (do_read)
when (enable : lead  (write eql 1))) := (do_write)
```

```
! ********************************************************************
!
!  Name      : IO.ISP
!  Purpose   : IO module used for class B SIC
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments : non-buffered IO module
!             this module only responds to io requests
!             and multiplies the last two data words
!             it received
!
! ********************************************************************

!
!  declarations
!

macro     ME         = 1 &,          ! my device name
          WORD       = 18 &,         ! basic word size
          STATUS     = 12 &,         ! status length
          BUSY       = 1 &,          ! busy
          DONE       = 0 &,          ! not busy
          CYCLE      = delay(1) &,   ! basic cycle time
          BIGCYCLE   = delay(5)&;    ! command cycle time

port      CSBUS  <STATUS>,           ! status bus
          IOBUS  <WORD>,             ! io data bus
          intline,                   ! interrupt line, not used
          csrdy,                     ! status ready
          ready,                     ! io ready
          datavalid,                 ! io data valid
          accept;                    ! io accepted

state     COMM <STATUS>,             ! SICs command
          STAT <STATUS>,             ! status register
          DATA <WORD>,               ! data register
          OLD_DATA <WORD);           ! old data register

format    io_dev     = COMM<11:9>,! io device
          io_command = COMM<8:8), ! command only
          io_data    = COMM<7:7>, ! data or status
          io_dir     = COMM<6:6>, ! io direction
          io_status  = COMM<5:0>; ! io status
```

```
! ***************************************************************
! receive data

receive :=
        (
         ready = 1;                          ! ready to send
         wait (datavalid : lead);            ! data on bus
         DATA = IOBUS; accept = 1; CYCLE;    ! get data
         wait (datavalid : trail);           ! ok
         ready = 0; accept = 0;              ! release bus
        )

! ***************************************************************
! send data

send :=
        (
         wait (ready : lead);                ! wait for ready
         case io_data
              0 : (IOBUS = DATA; CYCLE)      ! data on bus
              1 : (CSBUS = STAT; CYCLE)      ! status on bus
         esac;
         datavalid = 1;                      ! data valid
         wait (accept : lead);               ! wait til received
         datavalid = 0;
         case io_data                        ! release bus
              0 : (IOBUS = 0)
              1 : (CSBUS = 0)
         esac
        )

! ***************************************************************
! do something with data received

do_command :=
        (
         STAT = BUSY;                  ! set status to busy
         DATA = DATA * OLD_DATA;       ! perform multiply
         OLD_DATA = DATA;              ! and save old data
         BIGCYCLE;
         STAT = DONE;
        )
```

```
! ***************************************************************************
! main program

main :=
        (
         wait (csrdy : lead);
         COMM = CSBUS; accept = 1; CYCLE;
         accept = 0;

         if (io_dev eql ME)
            (
             case io_command
                  0 : (case io_dir
                       0 : (receive; do_command)
                       1 : (send)
                       esac)
                  1 : (do_command)
             esac
            )
        )
```

APPENDIX F

"B" MODEL TOPOLOGY FILE

The following code illustrates the topology file used to define the "B" level model SIC network.

```
! ********************************************************
!
!  Name      : BSIC.T  (topology file)
!  Purpose   : topology file for a
!              Small Instruction set Computer
!              class C implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
! ********************************************************

signals  intline <8>,              ! interrupt lines
         bcrdy   <4>,              ! buffer channel ready
         bufrdy  <4>,              ! buffer ready
         start,                    ! start signal
         csrdy,                    ! status ready
         ready,                    ! io ready
         datavalid,                ! io data valid
         accept,                   ! io accepted
         bufend,                   ! buffer done
         iobus  <18>,              ! iobus
         csbus  <12> ,             ! status bus
         mabus  <13>,              ! memory address bus
         mwrite,                   ! memory write
         menable;                  ! memory enable


! sic
```

```
processor sic = "bsic.sim";
   time delay 200ns;
   connections
         intline  = intline,
         bcrdy    = bcrdy,
         bufrdy   = bufrdy,
         start    = start,
         csrdy    = csrdy,
         ready    = ready,
         datavalid= datavalid,
         accept   = accept,
         bufend   = bufend,
         iobus    = iobus,
         csbus    = csbus,
         mabus    = mabus,
         mwrite   = mwrite,
         menable  = menable;

!  program memory

processor pram = "ram.sim";
   time delay 50ns;
   connections addr   = mabus,
               data   = iobus,
               write  = mwrite,
               enable = menable;
   initial     m      = coreimage;

! an io process

processor io = "io.sim";
   time delay 200ns;
   connections csbus     = csbus,
               iobus     = iobus,
               intline   = intline<1:1>,
               csrdy     = csrdy,
               ready     = ready,
               datavalid = datavalid,
               accept    = accept;
```

APPENDIX G

"B" MODEL METAMICRO SOURCE FOR SIC

The following metaMicro source code is used for defining the executable instructions of the "B" level implementation of SIC.

```
! ***************************************************************
!
!  Name      : BSIC.METAMICRO
!  Purpose   : metaMicro assembly code generator for SIC
!              class B implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments :
!
! ***************************************************************

instr    I[1,1]<18> $                ! 1 word instruction of 18 bits

format   opcode      = I<17:15>,     ! operation
         addr_type   = I<14:13>,     ! address type
         addr        = I<12:0>,      ! address

         rot_dir     = I<13:13>,     ! rotation direction
         rot1        = I<12:12>,     ! rotate in event 1
         rot2        = I<7:7>,       ! rotate in event 1
         rot3        = I<3:3>,       ! rotate in event 1
         ev1_1       = I<11:10>,     ! event time 1 sub event 1
         ev1_2       = I<9:8>,       ! event time 1 sub event 2
         ev2         = I<6:4>,       ! event time 2 only event
         ev3_lt      = I<2:2>,       ! event time 3 less than 0
         ev3_eq      = I<1:1>,       ! event time 3 equal 0
         ev3_gt      = I<0:0>,       ! event time 3 greater than 0
```

```
        op_io_int   = I<14:12>,    ! operate, I/O, and interrupt
                                   ! opcodes

        io_device   = I<11:9>,     ! I/O device
        io_command  = I<8:7>,      ! I/O command
        io_status   = I<11:0>,     ! I/O status
        io_data     = I<7:7>,      ! I/O data or status
        io_dir      = I<6:6>,      ! I/O direction
        io_comp     = I<5:0>,      ! I/O status compare

        buf_io_chan = I<10:9>$     ! buffer I/O channel

macro

! constants

        direct    = 0 &,
        indirect  = 1 &,
        index_a   = 2 &,
        index_b   = 3 &,
        left      = 0 &,
        right     = 1 &,

! the basic op codes

        isz(a,m)  = opcode = 0; addr = a; mode(m) $ &,
        lac(a,m)  = opcode = 1; addr = a; mode(m) $ &,
        and(a,m)  = opcode = 2; addr = a; mode(m) $ &,
        tad(a,m)  = opcode = 3; addr = a; mode(m) $ &,
        jms(a,m)  = opcode = 4; addr = a; mode(m) $ &,
        dac(a,m)  = opcode = 5; addr = a; mode(m) $ &,
        jmp(a,m)  = opcode = 6; addr = a; mode(m) $ &,

        ral       = opcode = 7; rot_dir = left; rot1 = 1;
        rar       = opcode = 7; rot_dir = right; rot1 = 1;

        nop       = opcode = 7; ev1_1 = 0 $ &,
        stl       = opcode = 7; ev1_1 = 1 $ &,
        cll       = opcode = 7; ev1_1 = 2 $ &,
        hlt       = opcode = 7; ev1_1 = 3 $ &,

        sta       = opcode = 7; ev1_2 = 1 $ &,
        cla       = opcode = 7; ev1_2 = 2 $ &,
        cma       = opcode = 7; ev1_2 = 3 $ &,
```

```
        szl         = opcode = 7; ev2 = 1 $ &,
        dfa         = opcode = 7; ev2 = 2 $ &,
        dfb         = opcode = 7; ev2 = 3 $ &,
        dta         = opcode = 7; ev2 = 4 $ &,
        ina         = opcode = 7; ev2 = 7 $ &,
        dtb         = opcode = 7; ev2 = 6 $ &,
        inb         = opcode = 7; ev2 = 7 $ &,

        skp         = opcode = 7; ev3_eq = 1; ev3_gt = 1 $ &,
        skz         = opcode = 7; ev3_eq = 1; $ &,

        od(n)       = opcode = 7; op_io_int = 5; io_device = n;
                      io_command = 0; io_dir = 0 $ &,
        id(n)       = opcode = 7; op_io_int = 5; io_device = n;
                      io_command = 0; io_dir = 1 $ &,
        is(n)       = opcode = 7; op_io_int = 5; io_device = n;
                      io_command = 1; io_dir = 1 $ &,
        ob(n)       = opcode = 7; op_io_int = 5; io_device = n;
                      io_command = 2; io_dir = 0 $ &,
        ib(n)       = opcode = 7; op_io_int = 5; io_device = n;
                      io_command = 2; io_dir = 1 $ &,
        oc(n)       = opcode = 7; op_io_int = 5; io_device = n;
                      io_command = 3; io_dir = 0 $ &,


! address mode determination

        mode(m)     =
                      if 'm eql "d" then {addr_type = direct};
                      if 'm eql "i" then {addr_type = indirect};
                      if 'm eql "a" then {addr_type = index_a};
                      if 'm eql "b" then {addr_type = index_b}; &,

! some more advanced opcodes, mainly operate instructions

        aral(n)   = opcode = 7; rot_dir = left; rot(n) $ &,
        arar(n)   = opcode = 7; rot_dir = right; rot(n) $ &,
        acla      = opcode = 7; ev1_2 = 2; ev2 = 4 $ &,
        aclb      = opcode = 7; ev1_2 = 1; ev2 = 6 $ &,

        rot(n)    = if n = 1 then {rot1 = 1};
                    if n = 2 then {rot1 = 1; rot2 = 1};
                    if n = 3 then {rot1 = 1; rot2 = 1; rot3 = 1} &;
```

APPENDIX H

"C" MODEL ISP' SOURCE FOR SIC

The following ISP' source code represents the "C" level model for SIC. In this version, all facilities of SIC are modeled as seperate ISP' processors.

```
! ***************************************************************
!
!  Name      : ALU.ISP  (alu)
!  Purpose   : ISP' code for an alu, used in the
!              Small Instruction set Computer
!              class C implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments : 2 18 bit inputs are processed, yields 19 bit output
!
! ***************************************************************

!
!  declarations
!

macro     WORDIN    = 18 &,        ! input word size
          WORDOUT   = 19 &,        ! output word size
          ALUDELAY  = 50 &,        ! alu delay

!  meanings of alu function

          ALU_A     = 0 &,         ! out = ina
          ALU_B     = 1 &,         ! out = inb
          ALU_ABAR  = 2 &,         ! out = not ina
          ALU_BBAR  = 3 &,         ! out = not inb
```

```
        ALU_ADD   = 4 &,          ! out = ina + inb
        ALU_AND   = 5 &,          ! out = ina and inb
        ALU_RAL   = 6 &,          ! out = rotate ina left
        ALU_RAR   = 7 &,          ! out = rotate ina right

port    ALU_FUNC <3>,             ! alu command
        INA      <WORDIN>,        ! input a
        INB      <WORDIN>,        ! input b
        lf,                       ! link flag input
        OUT      <WORDOUT>;       ! output

! ***************************************************************
! combinatatorial part of alu

do_alu :=
        (
         delay(ALUDELAY);
         case alu_func
              ALU_A    : (out = 0 concat ina)
              ALU_B    : (out = 0 concat inb)
              ALU_ABAR : (out = 0 concat (not ina))
              ALU_BBAR : (out = 0 concat (not inb))
              ALU_ADD  : (out = ina + inb)
              ALU_AND  : (out = ina and inb)
              ALU_RAL  : (out = ina concat lf)
              ALU_RAR  : (out = lf  conca  ina)
         esac;
        )

! ***************************************************************
! main routines, when anything changes compute ne outputs

when (INA)      := (do_alu)
when (INB)      := (do_alu)
when (ALU_FUNC) := (do_alu)
```

```
! ***************************************************************
!
!  Name     : BUSCON.ISP
!  Purpose  : ISP' code for A,B, and O bus control, used in the
!             Small Instruction set Computer
!             class C implementation
!  Author   : BJ Patz
!  Version  : 1.0
!
!  Comments :
!
! ***************************************************************

macro    BMDELAY = 10 &;        ! delay

port     CON1 <3>,              ! a bus control
         CON2 <3>,              ! b bus control
         CON3 <6>,              ! obus control

         A_CON <8>,             ! a bus control
         B_CON <8>,             ! b bus control
         O_CON <16>,            ! o bus control

!
!  control code meanings
!

!    a0  0 on ABUS
!    a1  1 on ABUS
!    a2  all 1's on ABUS
!    a3
!    a4
!    a5  IR on the ABUS
!    a6  AC on ABUS
!    a7  BWC on ABUS

!    b0  0 on BBUS
!    b1  1 on BBUS
!    b2  all 1's on BBUS
!    b3
!    b4  MD on the BBUS
!    b5  IA on BBUS
!    b6  IB on BBUS
!    b7  PC on BBUS
```

```
!     o0  no op
!     o1  IR = OBUS
!     o2  IR <addr_part> = OBUS
!     o3
!     o4
!     o5  AC = OBUS
!     o6  If,AC = OBUS
!     o7  MD = OBUS
!     o8  IA = OBUS
!     o9  IB = OBUS
!     o10 PC = OBUS
!     o11 BWC = OBUS
!     o12 MA = OBUS
!     o13 CSR = OBUS
!     o14 INTR = OBUS and INTR
!     o15 MR = OBUS


! ***************************************************************************
!  do de mux process

do_con :=
        (
         delay(BMDELAY);

         A_CON := 0;
         B_CON := 0;
         C_CON := 0;

         A_CON<CON1> = 1;
         B_CON<CON2> = 1;
         O_CON<CON3> = 1;

! ***************************************************************************
!  main

when (CON1) := (do_con)
when (CON2) := (do_con)
when (CON3) := (do_con)
```

```
! ***************************************************************
!
!  Name      : CCGEN.ISP  (condition code generator)
!  Purpose   : ISP' code for the condition code generator
!              Small Instruction set Computer
!              class C implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments : 1 8 bit mux, 1 32 bit mux
!
! ***************************************************************

!
!  declarations
!

MACRO    CCDELAY = 10 &;  !condition code delay

port     INA<8>,          ! A input lines
         INB<32>,         ! B input lines
         COND <12>        ! condition code select from u machine
         cc;              ! condition code
!
! condition code meanings
!

!  A bit 0,  nop
!  A bit 1,  accept
!  A bit 2,  datavalid
!  A bit 3,  ready
!  A bit 4,  IR * BCR
!  A bit 5,  BCR <CC>
!  A bit 6,  BIOR <CC>
!  A bit 7,
!  A bit 0,  true
!  B bit 1,  v(BCR)
!  B bit f
!  B bit 3,  status = 0
!  B bit 4,  status < 0
!  B bit 5,  status > 0
!  B bit 6,
!  B bit 7,  If
!  B bit 8,  f
```

```
!  B bit 9,  IR<17> * IR<16> * IR<15>
!  B bit 10, IR<17> * IR<16>
!  B bit 11, IR<2>
!  B bit 12, IR<3>
!  B bit 13, IR<4>
!  B bit 14, IR<5>
!  B bit 15, IR<6>
!  B bit 16, IR<7>
!  B bit 17, IR<8>
!  B bit 18, IR<9>
!  B bit 19, IR<10>
!  B bit 20, IR<11>
!  B bit 21, IR<12>
!  B bit 22, IR<13>
!  B bit 23, IR<14>
!  B bit 24, IR<15>
!  B bit 25, IR<13>
!  B bit 26, IR<17>
!  B bit 27,
!  B bit 28,
!  B bit 29,
!  B bit 30,
!  B bit 31,

! ***************************************************************
! cc generator

do_cc :=
        (
         delay(CC_DELAY);
         cc = COND<8:8> xor (INA<COND<7:3>> or INB<COND<2:0>>)
        )


! ***************************************************************
! main routines, when anuything changes compute cc

when (INA)   := (do_cc)
when (INB)   := (do_cc)
when (COND)  := (do_cc)
```

```
! ***************************************************************
!
!  Name      : CLKGEN.ISP  (clock generator)
!  Purpose   : ISP' code for a clock signal generator,
!              Small Instruction set Computer
!              class C implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
! ***************************************************************

!
!  declarations
!

macro  HITIME  = delay(150); ! time at 1
       LOWTIME = delay(50);  ! time at 0

port   clk;                  ! clock output

! ***************************************************************
! main program

main :=
       (
        clk = 1; HITIME;
        clk = 0; LOWTIME;
       )
```

```
! ****************************************************************
!
!  Name      : CSGEN.ISP  (control signal generator)
!  Purpose   : ISP' code for the control signal genrator
!              Small Instruction set Computer
!              class C implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments : 2 16 bit demuxes
!
! ****************************************************************

!
!  declarations
!

macro    CSDELAY = 10 &,          ! control delay

port     CONT <9>,                 ! 9 control bits
         OUTA <16>,                ! output signal A
         OUTB <16>;                ! output signal B

!
!  control signal meanings
!
!  A bit 0   no op
!  A bit 1   mem write
!  A bit 2   mem enable
!  A bit 3   accept = 1
!  A bit 4   datavalid = 1
!  A bit 5   datavalid = 0
!  A bit 6   ready = 1
!  A bit 7   bufend = 1
!  A bit 8
!  A bit 9
!  A bit 10  IOBUS = MD
!  A bit 11  CSBUS = CSR
!  A bit 12
!  A bit 13
!  A bit 14
!  A bit 15
!  B bit 0   no op
!  B bit 1   CSR = CSBUS
```

```
!  B bit 2   MD  = IOBUS
!  B bit 3   BIOR <CC> = 1
!  B bit 4   BCR  <CC> = 0
!  B bit 5   BCR  <CC> = 1
!  B bit 6   BUFRDY<CC> = 1
!  B bit 7   CC = CC+1
!  B bit 8    intf = 0
!  B bit 9   enif = 0
!  B bit 10  enif = 1
!  B bit 11   If = 0
!  B bit 12   If = 1
!  B bit 13   If = OBUS<0>
!  B bit 14   If = OBUS<18>
!  B bit 15   intf = ((v(MR * INTR)*enif)

! ***************************************************************
! de mux code

do_con :=
         (
          delay(CSDELAY);
          OUTA = 0;
          OUTB = 0;
          OUTA<CONT<3:0>> = 1;
          OUTA<CONT<7:4>> = 1;

! set enable if write

          if (CONT<3:0> eql 1) (OUTA<2> = 1);
         )

! ***************************************************************
! main routines

when (CONT) := (do_con)
```

```
! ***************************************************************
!
!  Name      : IOHANDLE.ISP  (io handler)
!  Purpose   : ISP' code for io for the
!              Small Instruction set Computer
!              class C implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments :
!
! ***************************************************************

macro     INT_CHAN  = 8 &,          ! number of interrupt lines
          STATUS    = 12 &;         ! size of status word

port      clk,                      ! clock
          INTLINE <INT_CHAN>,       ! interrupt lines
          CSBUS   <STATUS>,         ! status bus
          csrdy,                    ! status ready
          IN      <STATUS>,         ! obus connection
          c1 <4>,                   ! control 1
          c2 <8>,                   ! control 2
          accept,                   ! accept
          datavaild,                ! datavalid
          ready,                    ! ready
          inter_rcvd;               ! interrupt received
!
!  control meanings

!  c2 bit 0  intf = 0
!  c2 bit 1  enif = 0
!  c2 bit 2  enif = 1
!  c2 bit 3  INTR = not (INTR and IN)
!  c2 bit 4  MR = IN
!  c2 bit 5  intf =((INTR and MR) neq 0) and enif
!  c2 bit 6  CSBUS = CSR,csrdy = 1
!  c2 bit 7  CSR = CSBUS

!  c1 bit 0   accept = 1
!  c1 bit 1   datavalid = 1
!  c1 bit 2   datavalid = 0
!  c1 bit 3   ready = 1
```

```
state    MR   <INT_CHAN>,          ! mask register
         INTR <INT_CHAN>,          ! interrupt register
         CSR  <STATUS>,            ! status
         intf,                     ! intf
         enif;                     ! enif


! ***************************************************************
!  register ops

do_int :=
        (
         if c2<0> (intf = 0);
         if c2<1> (enif = 0);
         if c2<2> (enif = 1);
         if c2<3> (INTR = not (INTR and IR));
         if c2<4> (MR   = IN);
         if c2<5> (intf = ((INTR and MR) neq 0) and enif);
         if c2<6> (CSR = CSBUS);
         next;

         inter_rcvd = intf;
        )

! ***************************************************************
!  main processes

when (intline : lead) (INTR = INTR or intline)

when(clk : trail)  (do_int)

when (c2<7> : lead)   (CSBUS = CSR; csrdy = 1)
when (c1<0>)          (accept = c1<0>9)
when (c1<1> : lead)   (datavalid = 1)
when (c1<2> : lead)   (datavalid = 0)
when (c1<3>)          (ready  = c1<3>)
```

```
! ***************************************************************
!
!  Name      : LF.ISP  (link flag)
!  Purpose   : ISP' code for the link flag for a
!              Small Instruction set Computer
!              class C implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
! ***************************************************************

!
!  declarations
!

macro     REGSIZE  = 1 &,          ! register size
          REGDELAY = 10 &;         ! register delay

port      clk,                     ! clock
          c<4>                     ! control

! bit 0 clear link
! bit 1 set link
! bit 2 link = in1
! bit 3 link = in2

          in1,in2,                 ! input bits
          out                      ! outputs

state     R;                       ! register

! ***************************************************************
!  register output

do_link :=
          (
           if c<0> (R = 0);        ! clear link
           if c<1> (R = 1);        ! set link
           if c<2> (R = in1);      ! low bit of obus
           if c<3> (R = in2);      ! hi  bit of obus

           delay (REGDELAY);
           out = R;
          )
```

```
! ***************************************************************************
!  main routines

when (clk : trail) := (do_link)
```

```
! ***************************************************************
!
!  Name     : REG13.ISP  (13 bit register)
!  Purpose  : ISP' code for a 13 bit register, used in the
!             Small Instruction set Computer
!             class C implementation
!  Author   : BJ Patz
!  Version  : 1.0
!
!  Comments : register loads on negative edge
!             register has enable
!             register has output enable
!
! ***************************************************************

!
!  declarations
!

macro    REGSIZE  = 13 &,          ! register size
         REGDELAY = 15 &;          ! register delay

port     clk,                      ! clock
         en,                       ! enable
         oe,                       ! output enable
         IN  <REGSIZE>,            ! inputs
         OUT <REGSIZE>;            ! outputs

state    R   <REGSIZE>;            ! register

! ***************************************************************
!  register output

do_output :=
         (
          if oe delay(REGDELAY);
          case oe
             1 : (OUT = R)         ! output enabled
             0 : (OUT = 0)         ! release output
          esac
         )

! ***************************************************************
!  register input
```

```
do_input := (if en (R = IN))      ! input enabled

! ***************************************************************
!  main routines

when (clk : trail) := (do_input; do_output)
when (oe)          := (do_output)
```

```
! ***************************************************************
!
!  Name     : REG18.ISP  (18 bit register)
!  Purpose  : ISP' code for a 18 bit register, used in the
!             Small Instruction set Computer
!             class C implementation
!  Author   : BJ Patz
!  Version  : 1.0
!
!  Comments : register loads on negative edge
!             register has enable
!             register has output enable
!
! ***************************************************************

!
!  declarations
!

macro    REGSIZE  = 18 &,          ! register size
         REGDELAY = 15 &;          ! register delay
           .
port     clk,                      ! clock
         en,                       ! enable
         oe,                       ! output enable
         IN  <REGSIZE>,            ! inputs
         OUT <REGSIZE>;            ! outputs

state    R   <REGSIZE>;            ! register

! ***************************************************************
!  register output

do_output :=
         (
          if oe delay(REGDELAY);
          case oe
             1 : (OUT = R)         ! output enabled
             0 : (OUT = 0)         ! release output
          esac
         )

! ***************************************************************
!  register input
```

```
do_input := (if en (R = IN))      ! input enabled

! ***************************************************************************
!  main routines

when (clk : trail) := (do_input; do_output)
when (oe)          := (do_output)
```

```
! ***************************************************************
!
!  Name     : REG48.ISP  (48 bit register)
!  Purpose  : ISP' code for a 48 bit register, used in the
!             Small Instruction set Computer
!             class C implementation
!  Author   : BJ Patz
!  Version  : 1.0
!
!  Comments : register loads on positive edge of clock
!
! ***************************************************************

!
!  declarations
!

macro    REGSIZE = 48 &,          ! register size
         REGDELAY = 15 &;         ! register delay

port     clk,                     ! clock
         IN <REGSIZE>,            ! input to register
         OUT <REGSIZE>;           ! output of register

!
!  register bit field meanings
!

!  47:39  -  next address
!  38:27  -  next address control
!  26:24  -  unused
!  23:21  -  a bus control
!  20:18  -  b bus control
!  17:12  -  o bus control
!  11:9   -  alu func
!  8:0    -  misc control

state    R <REGSIZE>;

! ***************************************************************
!  register output

do_output :=
         (
          delay(REGDELAY);
```

```
        OUT = R;
       )

! ***************************************************************************
!  main routines

when (clk : lead) := (R = IN; do_output)
```

```
! ***************************************************************
!
!  Name      : REGIR.ISP
!  Purpose   : ISP' code for an 18 bit register,
!              and some logic (IR register)  used in the
!              Small Instruction set Computer
!              class C implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments : register loads on negative edge
!             register has enable
!             register has output enable
!
! ***************************************************************

!
!  declarations
!

macro     REGSIZE  = 18 &,          ! register size
          REGDELAY = 15 &;          ! register delay

port      clk,                      ! clock
          en1,                      ! enable all bits
          en2,                      ! enable only low 13 bits
          oe,                       ! output enable
          IN  <REGSIZE>,            ! inputs
          OUT <REGSIZE>,            ! outputs
          CONT<18>;                 ! IR bits for control

!
! control bit meaning
!

!  bit x    IR<x>
!  bit 1    IR<17> * IR<16>
!  bit 0    IR<17> * IR<16> * IR<15>

state     R <REGSIZE>;              ! register

! ***************************************************************
!  register output
```

```
do_output :=
          (
           if oe delay(REGDELAY);
           case oe
              1 : (OUT = R)          ! output enabled
              0 : (OUT = 0)          ! release output
           esac
          )

! ***************************************************************************
!  register load, and special output

do_input :=
         (
          if en1 (R       = IN);
          if en2 (R<12:0> = IN);
          next;
          if (en1 or en2)
                (
                 CONT      = R; next;
                 CONT<0:0> = R<17> and R<16> and R<15>;
                 CONT<1:1> = R<17> and R<16>;
                )
         )

! ***************************************************************************
!  main routines

when (clk : trail) := (do_input; do_output)
when (oe)          := (do_output)
```

```
! ***************************************************************
!
!  Name      : REG13.ISP  (13 bit register)
!  Purpose   : ISP' code for a 13 bit register,
!              MA register, used in the
!              Small Instruction set Computer
!              class C implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments : register loads on negative edge
!             register has enable
!
! ***************************************************************

!
!  declarations
!

macro     INSIZE   = 19 &,          ! all registers input
                                    ! from 19 bit bus
          OUTSIZE  = 13 &,          ! and output
                                    ! to 18 bit bus
          REGSIZE  = 13 &,          ! register size
          REGDELAY = 15 &;          ! register delay

port      clk,                      ! clock
          en,                       ! enable
          IN  <INSIZE>,             ! inputs
          OUT <OUTSIZE>;            ! outputs

state     R   <REGSIZE>;            ! register

! ***************************************************************
!  register output

do_output :=
          (
           delay(REGDELAY);
           OUT = R;
          )

! ***************************************************************
!  register input
```

```
do_input :=
         (
          if en (R = IN)                    ! input enabled
         )

! ***************************************************************************
!  main routines

when (clk : trail) := (do_input; do_output)
```

```
! ***************************************************************
!
!  Name     : REGMD.ISP  (18 bit register)
!  Purpose  : ISP' code for an 18 bit register,
!             and some misc logic (MD register), used in the
!             Small Instruction set Computer
!             class C implementation
!  Author   : BJ Patz
!  Version  : 1.0
!
!  Comments : register loads on negative edge
!             register has enable
!             register has output enable
!
! ***************************************************************

!
!  declarations
!

macro    REGSIZE  = 18 &,          ! register size
         REGDELAY = 15 &;          ! register delay

port     clk,                      ! clock
         en1,                      ! enable input obus
         en2,                      ! enable input iobus
         oe1,                      ! output enable abus
         oe2,                      ! output enable iobus
         IN1  <REGSIZE>,           ! obus input
         IN2  <REGSIZE>,           ! iobus input
         OUT1 <REGSIZE>,           ! abus output
         OUT2 <REGSIZE>,           ! iobus output

state    R <REGSIZE>;              ! register

! ***************************************************************
!  register output

do_output :=
         (
          if (oe1 or oe2) (delay(REGDELAY));
          case oe1
             1 : (OUT1 = R)        ! output enabled
             0 : (OUT1 = 0)        ! release output
```

```
          esac;
          case oe2
             1 : (OUT2 = R)          ! output enabled
             0 : (OUT2 = 0)          ! release output
          esac;
         )

! ***************************************************************************
!  register input

do_input :=
         (
          if en1        (R = IN1)
          else (if en2 (R = IN2))
         )

! ***************************************************************************
!  main routines

when (clk : trail) := (do_input; do_output)
when (oe)          := (do_output)
```

```
! ********************************************************************
!
!  Name      : ROM.ISP  (micro program rom)
!  Purpose   : ISP' code for an ROM, used in the
!              Small Instruction set Computer
!              class C implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments : 9 bit address, 48 bit data
!
! ********************************************************************

!
!  declarations
!
macro     DATASIZE = 48 &,          ! word size
          ADDRSIZE = 9 &,           ! address size
          ROMDELAY = 50 &;          ! rom delay

port      ADDR <ADDRSIZE>,          ! address
          DATA <DATASIZE>;          ! data

memory    ROM [0:511] <DATASIZE>; ! rom

! ********************************************************************
!  get data when address changes

when (ADDR) := (delay(ROMDELAY); DATA = ROM[ADDR])
```

```
! ***************************************************************
!
!  Name      : USEQ.ISP (micro sequencer)
!  Purpose   : ISP' code for a micro sequencer for a
!              Small Instruction set Computer,
!              class C implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
!  Comments : sequencer has internal memory of last address
!
! ***************************************************************

!
!  declarations
!

macro     ADDRSIZE  = 9 &,          ! address size
          SEQDEALY  = 50 &;         ! sequencer delay

port      clk,                      ! clock
          IN  <ADDRSIZE>,           ! next address input
          OUT <ADDRSIZE>,           ! address to rom
          branch;                   ! out = in if branch = 1

state     LASTADDR <ADDRSIZE>;      ! last address + 1

! ***************************************************************
! micro sequencer increment

do_seq_inc :=
          (
           LAST_ADDR = OUT + 1;
          )

! ***************************************************************
! micro sequencer generate address

do_seq :=
          (
           delay(SEQ_DEALY);
           case branch
                0 : (OUT = LAST_ADDR)
                1 : (OUT = IN)
           esac
```

```
        )

! ***************************************************************************
! main routines

when (clk : trail)          := (do_seq_inc)
when (clk : lead)           := (do_seq)
when (IN (clk eql 1))       := (do_seq)
when (branch (clk eql 1)) := (do_seq)
```

APPENDIX I

"C" MODEL TOPOLOGY FILE

The following code illustrates the topology file used to define the "C" level model SIC network.

```
! ********************************************************************
!
!  Name      : CSIC.T  (no pun intended, topology file)
!  Purpose   : topology file for a
!              Small Instruction set Computer
!              class C implementation
!  Author    : BJ Patz
!  Version   : 1.0
!
! ********************************************************************

signal   clk,                    ! clock
         romaddr  <9>,           ! rom address
         romdata  <48>,          ! rom data
         pipe     <48>,          ! pipeline register

         mabus    <13>,          ! program memory address bus
         abus     <18>,          ! abus
         bbus     <18>,          ! bbus
         obus     <19>,          ! obus
         iobus    <18>,          ! iobus
         lf,                     ! link flag

         intline  <8>,           ! interrupt lines
         start,                  ! start signal
         csrdy,                  ! status ready
         ready,                  ! io ready
         datavalid,              ! io data valid
         accept,                 ! io accepted
```

```
        csbus   <12>,           ! status bus
        intf,                   ! interrupt received
        int,                    ! single interrupt line

        a_con   <8>,            ! A bus control
        b_con   <8>,            ! B bus control
        o_con   <16>,           ! O bus control

        cont1   <16>,           ! misc control
        cont2   <16>,           ! misc control

        cond1   <8>,            ! condition code
        cond2   <32>,           ! condition code
        cc;                     ! branch condition

! ***************************************************************
! clock generator

processor clock = "clkgen.sim";
   time delay 1ns;
   connections clk    = clk;

! ***************************************************************
!  pipeline register

processor pipe = "reg48.sim";
   time delay 1ns;
   connections clk    = clk,
               in     = romdata,
               out    = pipe;

! ***************************************************************
!  microprogram rom

processor urom = "rom.sim";
   time delay 1ns;
   connections addr = romaddr,
               data = romdata;
   initial     rom  = coreimage;

! ***************************************************************
!  micro sequencer

processor useq = "useq.sim";
   time delay 1ns;
```

```
   connections clk    = clk,
               in     = pipe <47:39>,
               out    = romaddr,
               branch = cc;

! ***************************************************************
!  ma register

processor ma = "reg13.sim";
   time delay 1ns;
   connections clk    = clk,
               en     = o_con <12:12>,
               oe     = Hl,
               in     = obus  <17:0>,
               out    = mabus;


! ***************************************************************
!  md register

processor md = "regmd.sim";
   time delay 1ns;
   connections clk    = clk,
               en1    = o_con <7:7>,
               en2    = cont2 <2:2>,
               oe1    = b_con <4:4>,
               oe2    = cont1 <10:10>,
               in     = obus  <17:0>,
               out    = bbus;

! ***************************************************************
!  program memory

processor pram = "ram.sim";
   time delay 50ns;
   connections addr   = mabus,
               data   = iobus,
               write  = cont1<1:1>,
               enable = cont1<2:2>,
   initial     m  = coreimage;

! ***************************************************************
!  ir register

processor ir = "regir.sim";
```

```
   time delay 1ns;
   connections clk    = clk,
               en1    = o_con <7:7>,
               en2    = cont2 <2:2>,
               cont   = cond2 <26:9>,
               in     = obus  <17:0>,
               out    = abus;

! ***************************************************************
!  ac

processor ac = "reg18.sim";
   time delay 1ns;
   connections clk    = clk,
               en     = o_con <5:5>,
               oe     = a_con <6:6>,
               in     = obus  <17:0>,
               out    = abus;

! ***************************************************************
!  ia

processor ia = "reg13.sim";
   time delay 1ns;
   connections clk    = clk,
               en     = o_con <8:8>,
               oe     = b_con <5:5>,
               in     = obus  <17:0>,
               out    = bbus;

! ***************************************************************
!  ib

processor ib = "reg13.sim";
   time delay 1ns;
   connections clk    = clk,
               en     = o_con <9:9>,
               oe     = b_con <6:6>,
               in     = obus  <17:0>,
               out    = bbus;

! ***************************************************************
!  pc
```

```
processor pc = "reg13.sim";
   time delay 1ns;
   connections clk    = clk,
               en     = o_con <10:10>,
               oe     = b_con <7:7>,
               in     = obus  <17:0>,
               out    = bbus;


! ***************************************************************
!  alu

processor alu = "alu.sim";
   time delay 1ns;
   connections ina = abus,
               inb = bbus,
               lf  = lf,
               out = obus,
               alu_func = pipe <11:9>;

! ***************************************************************
! lf

processore lfp = "lf.sim";
   time delay 1ns;
   connections clk = clk,
               c   = cont2 <14:11>,
               in1 = obus  <0:0>,
               in2 = obus  <18:18>,
               out = lf;

! ***************************************************************
! io handler

processore ioh = "iohandle.sim";
   time delay 1ns;
   connections clk        = clk,
               int        = intline,
               csbus      = csbus,
               csrdy      = csrdy,
               in         = obus  <11:0>,
               c2         = cont2 <15:8>,
               c1         = cont1 <6:3>,
               accept     = accept,
```

```
                datavaild  = datavalid,
                ready      = ready,
                inter_rcvd = intf;

! ***************************************************************
! an io process

processor io = "io.sim";
   time delay 200ns;
   connections csbus     = csbus,
               iobus     = iobus,
               intline   = int,
               csrdy     = csrdy,
               ready     = ready,
               datavalid = datavalid,
               accept    = accept;

! ***************************************************************
! bus connections

processor buscon = "buscon.sim";
   time delay 1ns;
   connections con1  = pipe <23:21>,
               con2  = pipe <20:18>,
               con3  = pipe <17:12>,
               a_con = a_con,
               b_con = b_con,
               o_con = o_con;

! ***************************************************************
! control code generator

processor ccgen = "ccgen.sim";
   time delay 1ns;
   connections cond    = pipe <38:27>,
               ina     = cond1,
               inb     = cond2,
               cc      = cc;

! ***************************************************************
! control signal generator

processor csgen = "csgen.sim";
   time delay 1ns;
   connections cont  = pipe<8:0>,
```

```
outa  = cond1,
outb  = cond2;
```

LIST OF REFERENCES

Drongowski, Paul J., Martinez, M., and Yatin, T. A Guide for Writting N.mPC Hardware Models. Cleveland, Ohio : Case Western Reserve University, 1984.

Druian, Roy L. "Functional Models for VLSI Design," 20th ACM/IEEE Design Automation Conference Proceedings. June 1983.

Hill, Frederick J. and Peterson, G. R. Digital Systems: Hardware Organization and Design. 2nd ed. New York : John Wiley and Sons, 1978.

Ordy, Greg M. and Rose, C. W. "The N.2 System," 20th ACM/IEEE Design Automation Conference Proceedings. June 1983.

Ordy, Greg, N.mPC : Ecologist User's Manual. Cleveland, Ohio : Case Western Reserve University, 1978.

Parke, Frederic I. "An Introduction to the N.mPC Design Environment," 16th ACM/IEEE Design Automation Conference Proceedings. June 1979.

Rogers, L. R. and Ordy G. M. The MetaMicro User's Manual, Version 3.1. Cleveland, Ohio : Case Western Reserve University, July 1980.

Straubs, Ralph, ISP' User's Manual. Cleveland, Ohio : Case Western Reserve University, 1978.