

A Microcode Emulator for Undergraduate Teaching

David Abramson
Department of Computer Science
Monash University
Clayton 3168
Victoria
Australia

ABSTRACT

This paper describes an educational microcode emulation system. The emulator allows a student to write microcode for a particular bit slice processor, and then to observe the effects of the code. An assignment is set in which the student must implement a small minicomputer. The emulator then determines whether a solution is correct, and gives an index to its efficiency. The emulator differs from others in its external representation of the processor architecture and machine state.

1. INTRODUCTION

This paper describes a microcode emulator which was developed at Monash University [1]. For some years the department of Computer Science has taught a unit on microcoded control. As with most units the course has included a small assignment, designed to consolidate the theory taught in lectures.

Over the past few years this exercise has been done on the department's HP2100A minicomputer [9,10]. Because this equipment is becoming unreliable and expensive to maintain, and also because of the difficulty of debugging microcode, an alternative was sought for 1983.

2. SOLUTIONS

Various schemes were considered, from building real microcoded engines to an emulator. The first choice was to use the microcode of a real processor. At present most undergraduate teaching is done on VAX 11/780's [6]. This approach was not chosen for two main reasons. First, it was felt that the VAX microcode structure is too complex to teach. Second, because of the VAX architecture, it is impossible to safely share the machine amongst microcode writers and normal users. Thus the microcode class would require a VAX for its own dedicated use, which was considered far too expensive.

The next best choice was to build a number of dedicated microcoded engines which may be driven by a mainframe. However, because the engines are connected to a host they are effectively invisible. Thus, there is no advantage in using a real processor.

The third choice was to build an emulator. This system can present the same interface (or better) to the student as the other schemes but has many more advantages, such as better hardware utilization, better development aids, better debugging aids, graphical representation of the machine, etc. Consequently an emulator was chosen.

3. OTHER EMULATORS

The use of an emulator to teach computer architecture is by no means new. Various emulators have been written which allow a programmer to enter microcode, and then to single step the code and observe the results [11,12,13,17]. These have been used extensively by both programmers and students. However, most of the existing emulators have provided a flat view of the architecture. The emulator is controlled by a command line, with key words and parameters. From this view the target architecture is completely invisible, as all that is seen is a meaningless command line prompt.

Our major aim is to produce an emulator which reminds the student of the original architecture by providing a two dimensional view. A user can then browse through the various elements of the processor observing the effects of specific micro instructions.

Some emulators have allowed the student (or teacher) to describe the architecture using an abstract language (such as ISP [5] or ADL [19]). Among other advantages, the approach allows the student to try implementing the same primitives on different processors. This type of emulator is also extremely useful in a development site, and allows an emulator to be constructed quickly when a new processor is designed. On the initial implementation we decided not to include this facility.

4. AN EMULATOR - DESIGN GOALS

The emulator had a number of major design goals.

- (1) It should be interactive, and allow the user to view the effects of the microcode on the machine state. The user should be able to modify the machine state at any time, and execute instructions either one at a time or in a block.
- (2) It should be realistic. In designing a system for teaching there is a tendency to oversimplify the environment, thus removing some common practical problems.
- (3) It should be robust. A student environment is one of the most hostile available. Most of this protection must be provided by the operating system, which is ultimately responsible for the system security policy. However, the emulator itself should not contain any loopholes which allow students to cheat at any assignment.
- (4) It should be user friendly, and provide help messages when a student becomes lost.
- (5) There should be useful error messages which allow the student to pinpoint the cause of the error. This means that the valuable time allocated for assignments is not wasted on unimportant issues, and can be spent on important concepts.
- (6) The student should see a realistic picture of the machine, rather than an array of meaningless numbers. The student should not have to enter cryptic keywords followed by many parameters simply to find out the state of the processor. The pictures should preferably correspond to the many diagrams found in microprogramming textbooks. This gives the emulator a two dimensional view rather than a linear view.
- (7) The emulator should be able to use both cheap and expensive terminals. The University has a number of terminals attached to its machines, most of which are simple cursor addressable VDU's and some of which provide line drawing capabilities and field attributes. If all of the features are present then the emulator should provide a realistic picture of the machine. If the terminal does not provide all of the features then the machine representation should be degraded, rather than the emulator functionality.

- (8) The emulator implementation should be machine independent. This allows it to be used at many colleges even if different machines and operating systems are available.
- (9) In order to simplify debugging the emulator should support breakpoint interrupts on both the control store and the main memory. In this way a block of working code may be executed, and the machine halted before a section of faulty code is reached.
- (10) The emulator should allow easy assignment testing so that a student can determine whether the code is correct, and if not where the errors are. This testing should also help the teacher in assessing the student's work.
- (11) Since it may be desirable to distribute a partly solved assignment, the teacher should be able to give students a block of microcode for inclusion in their solution.
- (12) It should be possible to load the control store, the main memory and the machine state (i.e. the register, the buses) when the emulator is started.

5. THE ARCHITECTURE

Since it was decided that the architecture of the emulated processor should be as realistic as possible, the arithmetic and logic unit is largely based on the AMD2901 bit slice chip [18]. These units are combined to form a sixteen bit data pathway. The processor contains eight 16 bit registers, which are accessed by three data buses, as shown in Figure 1. Eight registers were chosen because it was possible to display them across the VDU screen. Any fewer would have made it impossible to set a reasonable complex assignment, and any more would have crowded the display. Some of the registers have fixed roles: These are the program counter (P) which points to the next macro instruction; the macro instruction register (IR); the accumulator (ACC); the memory data register (MDR) which is used to hold the most recently read data, or data to be written to the memory; the memory address register (MAR) which holds the current memory address. The three other registers are used as general microcode scratchpads (SP1 - SP3). Two extra one bit registers are used to hold the carry and overflow state.

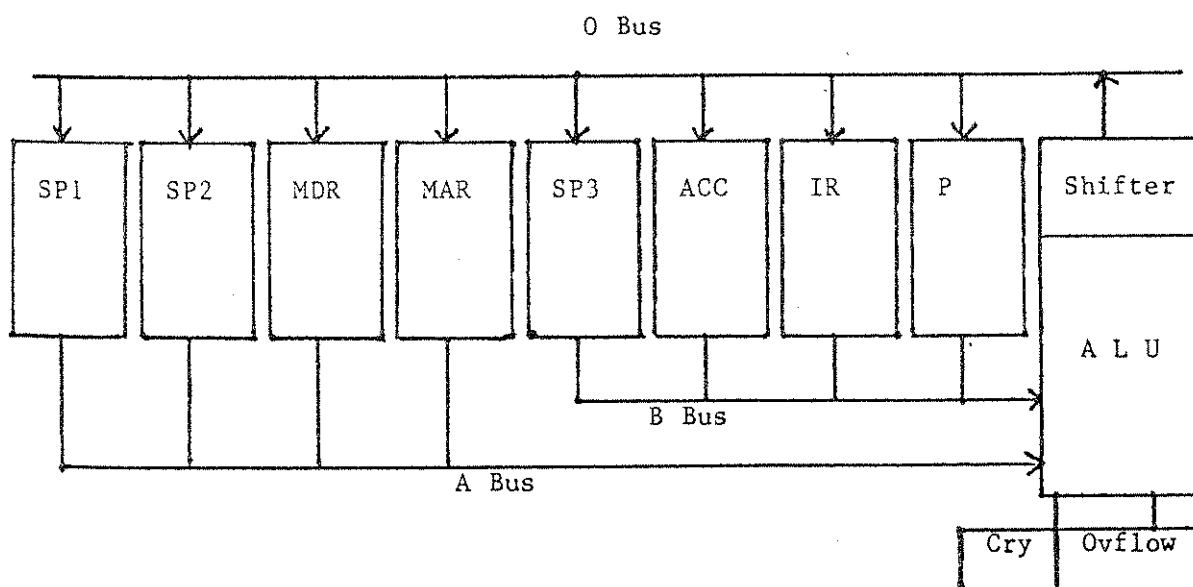


Figure 1 - The MICROEM Machine

The registers and data pathways are controlled by the microcoded control engine, consisting of a microprogram address register, a control store and a microprogram instruction register, as shown in Figure 2. Again, so that the controller was as realistic as possible it was based on the AMD2910 sequencer chips [18], although it has been simplified in some respects. Microcode is held in the writable control store, which can be loaded either directly through the emulator, or from a source file. The next control store address is derived from an address register, which is normally incremented, but is loaded during a microcode jump, jump to subroutine or return instruction. Microinstructions are read from the control store and placed in an instruction register. Each instruction is 40 bits in length, and is divided as shown in Figure 3. The A bus field controls which register is placed onto the A bus. The B bus field controls which register is placed onto the B Bus. The output bus field determines which register is clocked with the data on the output bus. The function field selects a valid arithmetic or logic function for the ALU, and also determines if a jump is to be executed. The shift field determines whether the shifter alters the data before it flows onto the output bus. The skip field selects a conditional skip, which bypasses the next instruction if true. The memory field is used for operating the memory unit. The target field is used for holding the target address if a jump is specified in the function field, or a 16 bit constant if one is specified in the A or B bus fields.

Microcode is executed in four distinct phases, without any pipelining, as shown in Figure 4. The first phase fetches the micro instruction from the control store and places it in the microinstruction register. From the second phase until the end of the instruction the A or B bus registers are placed onto the buses. From the third phase until the end of the instruction the ALU operates on the A and B buses and places data onto the output bus. Also during the third phase the main memory cycle is completed, providing data from a read, or using data from a write. During the fourth phase the data may be saved in a register, and the skip and jump instructions are executed. Whilst this sequence is uncommon and probably too slow to be used in any real machine, it was felt that it allowed more flexibility for teaching the concepts of data flow and buses. In this way the student can observe the data flowing out of the registers onto the buses, through the ALU and finally into a target register.

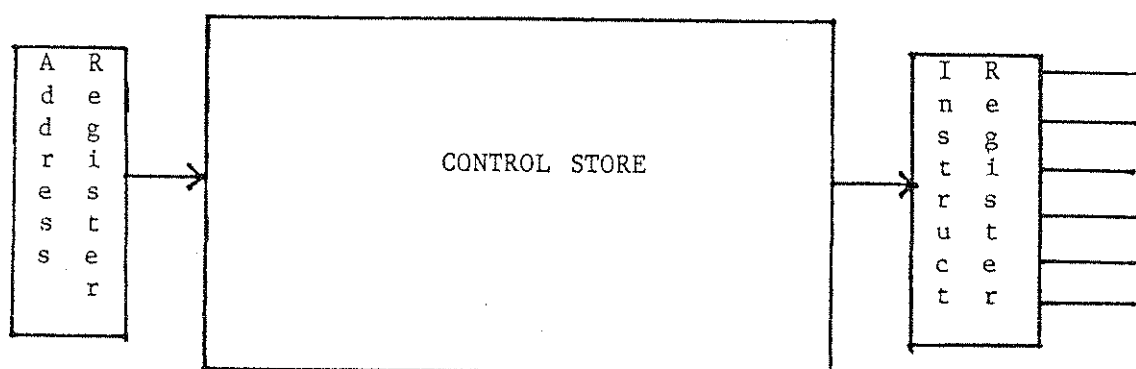


Figure 2 - The Control Unit

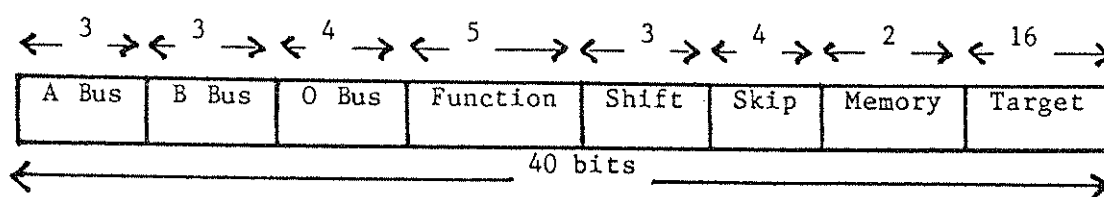


Figure 3 - the microinstruction

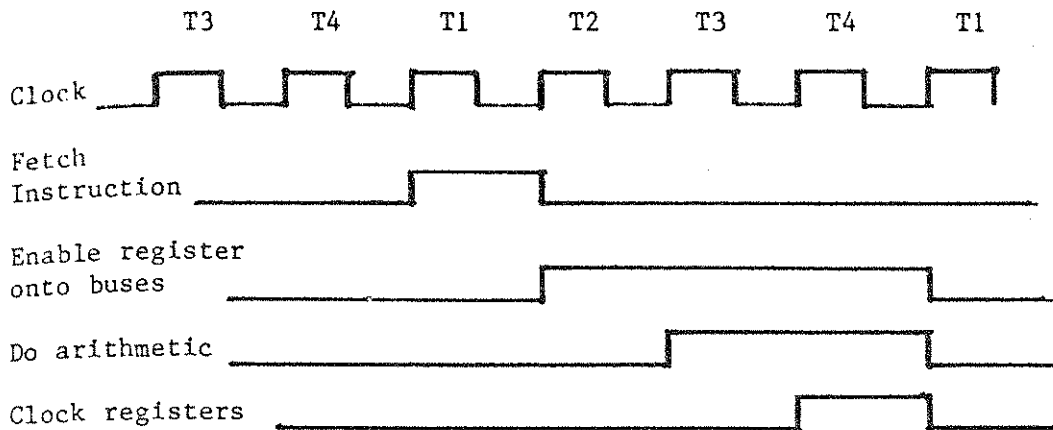


Figure 4 - the machine timing

The memory unit consists of 2048 16 bit words, and is accessed via the memory data and address registers, as shown in Figure 5. If a memory operation is requested, the contents of the memory address register is used as an index into the memory. If a write is requested then the data is taken from the memory data register and during a read data is returned to this register. The timing of the memory unit is invisible to the microcode, which is frozen if the memory unit takes longer than one micro instruction time to operate.

6. A USERS VIEW

6.1. Screen Choices

A user of the emulator is presented with a number of possible screens, as categorised below:

- Main Menu Screen
- Register Transfer Screen
- Main Memory Screen
- Control Store Screen
- Control Store Breakpoint Screen
- Main Memory Breakpoint Screen

A user may traverse these screens by using the terminal function keys. A number of other terminal keys are active at all times. These allow the user to tab (and backtab) to various fields of the screen, redraw the screen and ask for help. If more than one line of help is available then the entire screen is replaced by one or more help screens. To maintain terminal independence these keys may be redefined to be any character sequence.

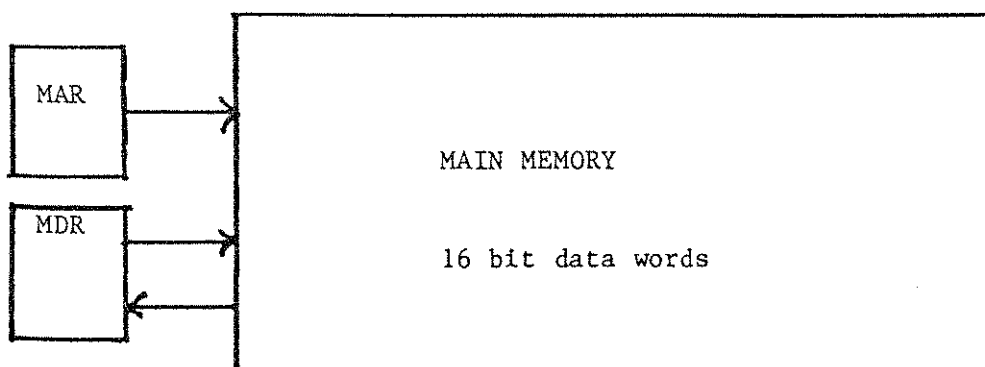


Figure 5 - The Main Memory Unit

6.2. The Main Menu

This screen is the first displayed after the emulator has been invoked. It asks for the name of a file from which microcode can be loaded, and allows the user to select any of the other screens. If the student ever returns to the main menu then control store is reloaded. Also, if there are files to initialize the main memory and the machine state (i.e. the register values) then these are used at this time.

6.3. The Register Transfer Screen

This screen displays a picture of the machine registers, buses, ALU, Shifter (as shown in Figure 1 and in the appendix) together with the current microinstruction. The user is allowed to modify any of the data values or the next microinstruction and execute the instructions one at a time, one phase of the instruction at a time or a block of instructions at a time. If the instruction is executed one phase at a time, the user may view the data flowing out onto the buses, through the ALU and onto the output bus and then finally being stored into a register. Special function keys allow any of the other screens to be selected. If the terminal has a line drawing capability then a realistic picture of the machine is shown.

6.4. Main Memory Screen

This screen displays the contents of the main memory as hexadecimal address and data pairs. Since the memory is too large to display on one screen, the user may select a window of 64 locations. The cursor may be moved to any location in the window and the contents modified.

6.5. Control Store Screen

This screen displays a window into the control store. Each microinstruction is displayed in its source form, and may be modified by moving the cursor to the correct field of the instruction.

6.6. Breakpoints

To allow breakpoints to be set on any control store reference and on any main memory reference two breakpoint tables are maintained. If the user wishes to detect when an address is referenced then the address value must be placed in a free location of the appropriate breakpoint table. The tables are manipulated by special screens, each of which holds up to 64 breakpoints. When a breakpoint is detected further interrupts are inhibited until the memory reference has been executed. After this the breakpoint is re-armed.

6.7. Testing some microcode

After a student has created and debugged a section of microcode it may be tested and compared against the correct answer. In this operation the emulator loads the control store with the student microcode, the main memory with a teacher initialization file and the machine state with a teacher initialization file. The microcode is then executed until an exception occurs (such as a halt micro instruction). The memory image is then compared against an answer memory file, and the machine state is compared against an answer state file. If either is incorrect, then the appropriate screen is selected and the erroneous information is tagged. For example, if a register value is incorrect, control is returned to the register transfer screen with the erroneous register value flashing. The student may then correct the microcode and retest the solution.

When the code is deemed correct a listing is generated which contains the code, a statement that the code has worked and some measure of code size and efficiency. This information is also written to a teacher log file which

may later be compared with the listing handed in by the student. In this way a student is discouraged from stealing a solution.

We have recently developed a marking program which takes the information in the log file and calculates a mark for each student based on the efficiency of the microcode and the size of the documentation.

7. AN ASSIGNMENT

Near the end of the lecture series students are set an assignment using the emulator. In previous years a small problem, such as writing microcode to perform a memory to memory move instruction, has been all which could be completed in the available time. This has been because there are so many other details that a user must know before they can write microcode for an HP2100 minicomputer. Also, code for these machines is notoriously hard to debug.

The emulator has allowed us to set a much more substantial problem, namely the emulation of a simple 16 bit minicomputer similar to machines such as the HP2100. In this case the teacher memory file contains a macro test program which exercises the student microcode.

8. FUTURE DIRECTIONS

At present the emulator is constructed from a section of special code, and is linked to a screen management package. In order to change the layout of the screens only the screen specification files need to be altered. However, if the machine behaviour is to be altered then the dedicated code must be changed. While the emulator is very highly modularized, modifying the code to emulate another architecture still involves a large amount of work (The original implementation took about two man months). An alternative approach is to provide a general emulator which interprets a machine description, such as those of [2,3,4,5,7,14,15,19]. This would greatly decrease the time required to produce a new emulator for a different architecture.

9. CONCLUSIONS

At the time of writing this paper one batch of students had just completed an assignment. The overall reaction to the emulator has been favorable; most students seem to have learnt more from this exercise and enjoyed the assignment more than students of previous years. From a class size of 90, most managed to successfully solve the problem.

The emulator has fulfilled the major design goals cited in section 3. The use of modern display terminals allows it to be interactive. The choice of the AMD bit slice chips means that it is fairly realistic. The system is robust in two respects. First, when a student tests a section of microcode the memory and processor state are initialized with files owned by the teacher. Similarly, the final machine state is compared with the teacher's files. Thus there is little possibility of cheating the emulator. Second, the screen management system filters out much of the erroneous data before the emulator is invoked. The use of both help messages and realistic error messages make the system user friendly. The machine independence is guaranteed by implementing the emulator code in the COBOL-74 language. Further, the screen functions are handled by a portable screen manager. Finally, the use of the display screens together with the line drawing capabilities of most modern terminals gives the user a two dimensional view of the architecture rather than the conventional linear view.

ACKNOWLEDGEMENTS

The author wishes to thank his 1982 project students for their part in the screen management system, and his 1983 third year students for being guinea pigs for the emulator. Thanks are also due to Ken McDonnell for reading an early draft of this paper, and also to the referees for their helpful comments.

REFERENCES

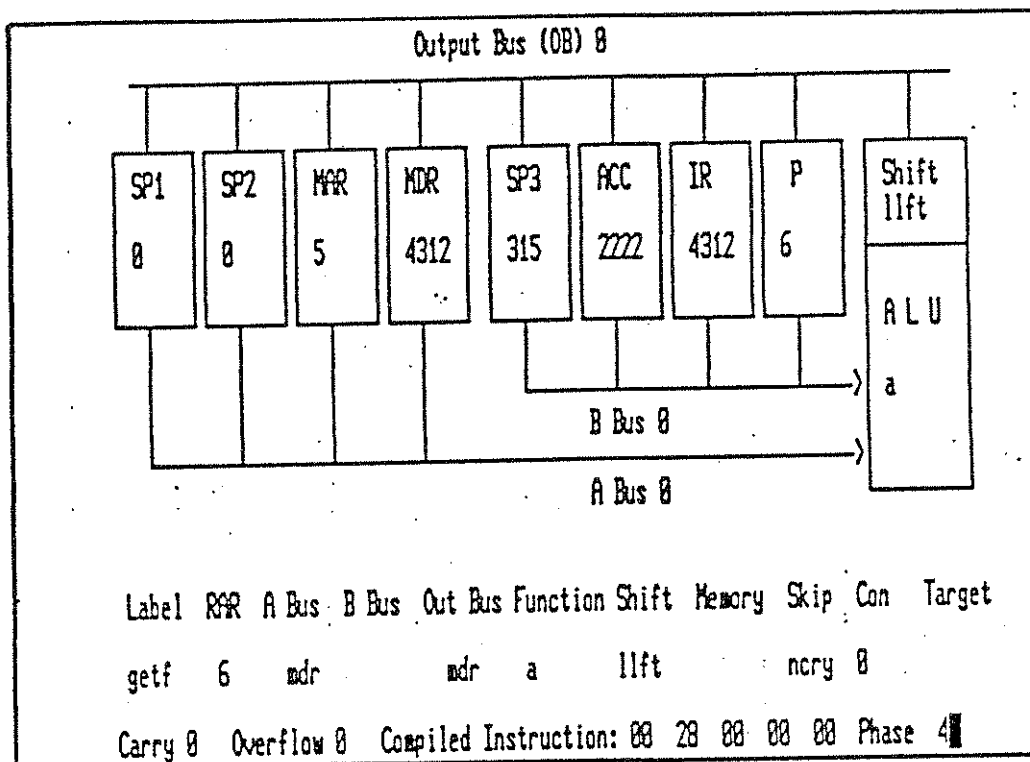
- [1] Abramson, D.A (1983) " A Microcode Emulator for Undergraduate Teaching - Users Manual", Monash University Technical Note 13, Department of Computer Science, Monash University.
- [2] Adamowicz, M. and Mirza, J. (1977) "MDSL: A Microcomputer Design and Simulation Language", SIG MICRO Newsletter, Vol 8, No 2, June 1977.
- [3] Ballieu G., Willems Y.D. and Lewi, J. (1981) "Effective Use of Microprogramming Capabilities", Seventh Euromicro Symposium on Microprogramming and Microprocessing, September 1981, pp 307-316.
- [5] Bell C.G and Newell A. (1970) "The PMS and ISP descriptive Systems for Computer Structures", AFIPS Spring Joint Computer Conference, pp 351-374.
- [6] Digital Equipment (1981) "The VAX Architecture Handbook".
- [7] Drongowski P.J. and Rose C.W. (1979) "Application of Hardware Description Languages to Microprogramming: Method, Practice and Limitations", 12th Annual ACM conference on Microprogramming, pp 55-59.
- [8] Fahrnich R. and Waldschmidt K. (1981) "Approach to Automatic Minimization and Realization of Microprogrammed Control Units" Seventh Euromicro Symposium on Microprogramming and Microprocessing, September 1981, pp 361-368.
- [9] Hewlett Packard (1972) "Microprogramming Guide for the HP2100A Minicomputer".
- [10] Hewlett Packard (1972) "Microprogramming Software for the HP2100A Minicomputer".
- [11] Lucindo, T., Chattergy, R., Pooch, U. (1981) "A Survey of Microprogram Verification and Validation Methods", The Computer Journal, Vol 24, No 2, May 1981, pp 139-142.
- [12] Mathur F.P. (1977) "*MICROSIM: A Microinstruction Simulator for Teaching Microprogramming and Emulation", 10th Annual ACM conference on Microprogramming, pp 109-118.
- [13] Mezzalma M. and Prinetto P. (1979) "Design and Implementation of a Flexible and Interactive Microprogram Simulator", 12th Annual ACM Workshop on Microprogramming, 1979 pp 42-48.
- [14] Meith W.H. and Richter L. (1981) "MMDS - A Microprogram Development Tool", Seventh Euromicro Symposium on Microprogramming and Microprocessing, September 1981, pp 261-268.
- [15] Mueller R.A. (1980) "Formalization and Automated Synthesis of Microprograms", 13th Annual ACM conference on Microprogramming, pp 45-53.
- [16] Nagle A.W. "Automatic Synthesis of Microcontrollers", 11th Annual ACM

conference on Microprogramming, pp 112-117.

- [17] Persson M. (1979) "Design of Software Tools for Microprogrammable Microprocessors", Fifth Euromicro Symposium on microprocessing and microprogramming, August 1979, pp 119-127.
- [18] Advanced Micro Devices (1978) "The AM2900 Family Data Book".
- [19] Fendnades (1980) "ADL: An Architectural Description Language", Internal Report TR80/22, Imperial College of Science and Technology, Department of Computing, London.

APPENDIX - SCREEN FORMATS

The following diagrams show some examples of the screens displayed by the emulator.



Main Memory Contents Start at: 0

Address	Data	Address	Data	Address	Data	Address	Data
1	0	2	4312	3	4314	4	5315
5	4312	6	1310	7	531c	8	4311
9	2313	a	531d	b	6050	c	a001
d	0	e	0	f	0	10	0
11	0	12	0	13	0	14	0
15	0	16	0	17	0	18	0
19	0	1a	0	1b	0	1c	0
1d	0	1e	0	1f	0	20	0
21	0	22	0	23	0	24	0
25	0	26	0	27	0	28	0
29	0	2a	0	2b	0	2c	0
2d	0	2e	0	2f	0	30	0
31	0	32	0	33	0	34	0
35	0	36	0	37	0	38	0
39	0	3a	0	3b	0	3c	0
3d	0	3e	0	3f	0	40	0

Control Store Contents Start at: 0

Label	R/R	A Bus	B Bus	Out Bus	Function	Shift	Memory	Skip	Con	Target
	1	mdr		ir	a				0	
	2	con	p	p	atb				1	
	3			sp1	zero				0	
test2	4	mdr		sp3	a				0	
	5	sp1	sp3		b-a-1			nneg	0	
	6				jmp				0	stloop
stloop	7	mdr		sp3	a				0	
loop	8	sp1		mar	a				0	
	9						read		0	
	a	sp2		mar	a				0	
getf	b	mdr		mdr	a	llft		ncry	0	
	c	sp1	con	sp1	atb				1	
	d	sp1			jidxa				0	exetab
getad	e	con	ir	sp3	b.a				7ff	
	f	con	ir		b.a			nzer	800	
	10				jmp				0	spare

AUSTRALIAN COMPUTER SCIENCE COMMUNICATIONS

Vol. 6, Number 1

February 1984

**PROCEEDINGS OF THE
SEVENTH AUSTRALIAN COMPUTER SCIENCE CONFERENCE**

Department of Computer Science,
University of Adelaide,
Adelaide, South Australia 5000

ACSC-7