Development of an 8-bit CPU using TTL logic

by

Muhamad Aidil b Jazmi, 3534

Dissertation submitted in partial fulfilment of
the requirements for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

DECEMBER 2006

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

Development of an 8-bit CPU using TTL logic

by

Muhamad Aidil b Jazmi

Dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfillment of the requirement for the
BACHELOR OF ENGINEERING (Hons)
(ELECTRICAL & ELECTRONICS ENGINEERING)

Approved by,

[illegible]

(Dr. Yap Vooi Voon)

UNIVERSITI TEKNOLOGI PETRONAS

TRONOH, PERAK

December 2006

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

[illegible]

MUHAMAD AIDIL B JAZMI

ABSTRACT

Computer is an integral part of human life nowadays and the complexity of computers grows in parallel with their processing capability. This project will build the basis of understanding the operation of the central processing unit of a computer by developing an 8-bit central processing unit from discrete TTL logic ICs. This CPU will also be used as a teaching aid for Computer System Architecture class in UTP. By building the CPU discretely, detailed operation of a computer can be understood from the hardware up to software level. The project discusses detailed electrical operation of blocks in the central processing unit mainly the processor. At the end of the project, a fully working microcomputer was constructed and studied in detail.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS AND NOMENCLATURES

CSA	computer system architecture
UTP	Universiti Teknologi Petronas
TLB	translation lookaside buffer
UART	universal asynchronous receiver transmitter
TTL	transistor-transistor logic
IC	integrated circuit
ROM	read only memory
PROM	programmable read only memory
ALU	arithmetic logic unit
CPU	central processing unit
DMA	direct memory access
CD	compact disc
RAM	random access memory
SRAM	static random access memory
IR	instruction register
MDR	memory data register
PTB	page table base
MSW	machine status word
PC	program counter
MAR	memory address register
MUX	multiplexer

CHAPTER 1
INTRODUCTION

To many end users, the internal workings parts of a CPU are difficult to comprehend. Computers accelerated economical development as they help to compute and process data in a way which is unachievable by humans. Societies are very dependent on computers that computers had become an essential part of human life. Due to this fact, computers are designed to be more helpful by cramping more features and increasing their capabilities.

1.1 Background

Computer which is controlled by the central processing unit had been already available for years. The exact timeline of computers can be traced back to 1950s by the invention of instruction list which basically list down the operations for an automated machine.

Today, computers like PCs and Macs are capable of delivering high resolution graphics and surround sound which has been taken for granted. How many of the students fully understand the different components like ALU, registers and the controls which are put together for a computer to function correctly? These machines are highly complex and it is a challenge to show how all the different components are assembled to form a functional computer.

1.2 Problem statement

In the view of the previous discussion, teaching Computer System Architecture (CSA) can be quite a challenge as it involves describing a lot of difficult technical details. Technical details in a computer systems course can be presented well by using a suitable teaching platform. This project describes the development of an 8-bit computer using TTL logic gates as a platform to support the teaching of CSA at Universiti Teknologi Petronas (UTP).

One of the goals of the CSA course is to explain the role and interaction of the components of a computer system therefore the teaching platform should have the following features.

1. A simple model architecture, with an easy to teach and learn instruction set
2. An architecture that can easily be used to demonstrate the relationship between different components of a computer system.
3. In addition, the platform should be able to provide the students the opportunity to learn the "ins" and "outs" of a computer system at gate level, which programming simulators does not [1].

A major problem in teaching computer architecture courses is how to help students make the cognitive leap that connects their theoretical knowledge with practical examples [2]. Different educators involved in teaching computer architecture and organization have to resolve this problem using a variety of computer system simulation software [2]. Although these simulators are useful, they however, still do not provide the students the "ins" and "outs" of a computer system.

Fundamentals	Organization of the CPU	Computer Arithmetic	Main Memory	Interfacing and Communication
▪ Registers and register file ▪ Data types ▪ Instruction types ▪ Addressing modes ▪ Instruction formats ▪ Fetch, decode, execution cycles	▪ Single vs multiple bus datapaths ▪ Pipelined, non-pipelined ▪ Control unit: hardwired vs. microprogrammed realization	▪ Representation of integers (signed, unsigned) ▪ Basic arithmetic algorithms for integer addition, subtraction, multiplication, and division ▪ Representation of real numbers ▪ Basic arithmetic algorithms for operations on real	▪ Memory hierarchies ▪ Main memory organization ▪ Latency, bandwidth, cycle time, performance ▪ Virtual memory system ▪ Cache memory ▪ Memory interleaving ▪ Memory technologies	▪ I/O fundamentals: handshaking, buffering ▪ I/O techniques: programmed I/O, interrupt driven, DMA ▪ Interrupt structures: vectored and prioritized, interrupt overhead, interrupts and re-entrant code. ▪ Buses: clock, control, address and data busses, arbitration

▪ I/O techniques and interrupt	▪ Arithmetic units implementation	numbers ▪ Conversions between real and integer numbers	(SRAM, DRAM, EPROM, Flash) ▪ Reliability and error correction	▪ Parallel and serial interfaces ▪ Timers

Table 1, Core topics in computer architecture and organization

1.3 Objective and scope of study

As mentioned earlier, the objective of this project is to develop an 8-bit computer using TTL gates as a platform to support teaching CSA in UTP. This project is relevant as it only requires basic knowledge of digital systems and microprocessors. A good working knowledge on digital circuits and practical electrical issues is required though because circuits in a discrete processor may become complicated and requires a lot debugging.

In time frame point of view, the project is viewed feasible as there is no major designing involved. The scope of the project is to build and debug the CPU until it works as intended and in the process attaining full comprehension of it.

Scope of the project:

- Build and test
  - The ALU and registers
  - The control and instruction sequencing circuit
  - Memory circuit
  - Clocks
  - Devices (Boot ROM, serial ports)
- Optional
  - Assembler and
  - C compiler

CHAPTER 2
LITERATURE REVIEW

A microprocessor executes a collection of machine instructions that tell the processor what to do. Based on the instructions, a microprocessor performs three basic operations.

Using its ALU (Arithmetic/Logic Unit), a microprocessor can perform mathematical operations like addition, subtraction, multiplication and division. Modern microprocessors contain complete floating point processors that can perform extremely sophisticated operations on large floating point numbers. A microprocessor can move data from one memory location to another and a microprocessor can make decisions and jump to a new set of instructions based on those decisions.

There may be very sophisticated things that a microprocessor does, but those are its three basic activities. Other than that, microprocessor comprises of registers as temporary storage area, buses to transport data and select memory areas and control lines to control all the blocks inside the microprocessor so that the instruction are executed correctly [3].

2.1 Page table

A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are those unique to the accessing process. Physical addresses are those unique to the CPU, i.e., RAM.

Say we have a computer architecture where the word size is 32 bits. This means we are able to form addresses from 0x00000000 to 0xffffffff - spanning 4GB. These addresses form what is called as the *virtual address space*. These addresses have no physical meaning - if we only have 16MB of memory, all addresses above 0x01000000 would be invalid. However, as mentioned, almost all programs do not use all 4GB of memory when a program runs, but only parts of it at a time. For example, the text, data, and stack segments may only be used and together only take 1 megabyte in total over the time where it runs.

The chunks as mentioned above are called special names. This 4GB virtual address space is split up into chunks, commonly 4K in size, called *pages*. The physical memory is also split up into chunks, also commonly 4K in size, called *frames*. A program's text segment might start at the virtual address 0x00000004 - page number 0x0, and offset 0x4, but in reality, this may correspond to the physical address 0xff0e0004 - frame number 0xff0e, and offset 0x4. What the virtual memory system does is convert virtual addresses into physical addresses, essentially, mappings between pages and frames. The page table is used for this purpose.

Many architecture also have direct hardware support for virtual memory, providing what is known as a translation lookaside buffer (TLB), which is filled with page-frame mappings initially, and instead of having the virtual memory system entirely in software, when the hardware looks up a memory address and does the page-frame translation, which gains us a performance increase.

However, the TLB can only hold a fixed number of page-frame mappings. It is the job of the virtual memory system to extend this into software, and to hold extra page-frame mappings. The virtual memory system does so by means of a page table [4].

2.1.1 Role of the page table

Assuming a program is running and it tries to access memory in the virtual address 0xd09fbabe. The virtual address is broken up into two: 0xd09f is the page number and 0xbabe is the offset, within the page 0xd09f.

With hardware support for virtual memory, the address is looked up within the TLB. The TLB is specifically designed to perform this lookup in parallel, so this process is extremely fast. If there is a match for page 0xd09f within the TLB (a TLB hit), the physical frame number is retrieved, the offset replaced, and the memory access can continue. However, if there is no match (called a TLB miss), the second port-of-call is the page table.

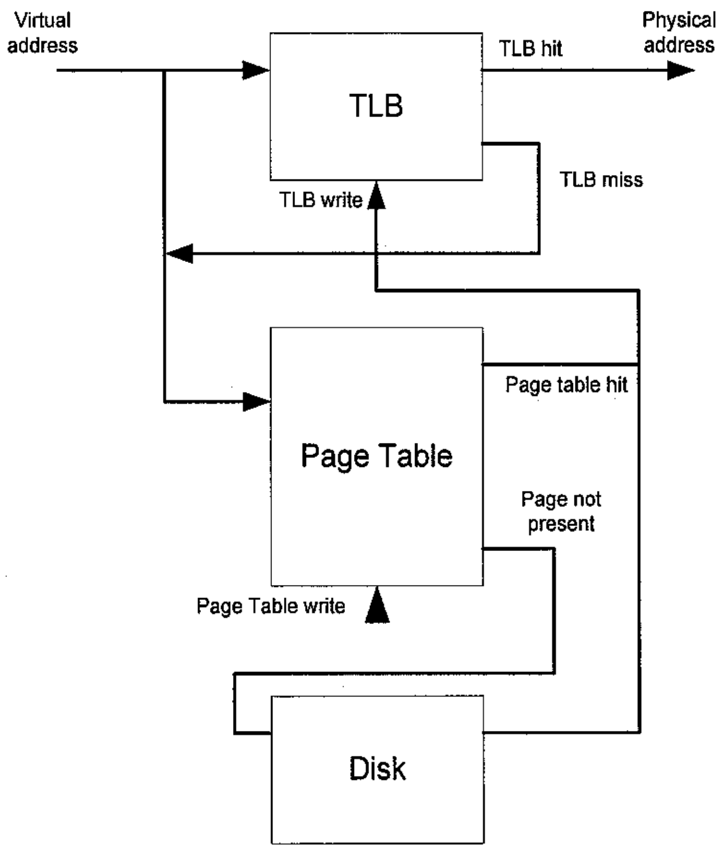


Fig. 1, Actions taken upon a virtual to physical address translation. Each translation is restarted if a TLB miss occurs, so that the lookup can occur correctly through hardware [4].

When the hardware is unable to find a physical frame for a virtual page, it will generate a processor interrupt called a page fault. Hardware architectures offer the chance for an interrupt handler to be installed by the operating system to deal with such page faults. The handler can look up the address mapping in the page table, and can see whether a mapping exists in the page table. If one exists, it is written back to the TLB, as the hardware accesses memory through the TLB in a virtual memory system, and the faulting instruction is restarted, with the consequence that the hardware will look in the TLB again, find the mapping, and the translation will succeed.

However, the page table lookup may not be successful for two reasons:

- there is no translation available for that address - the memory access to that virtual address is thus bad or invalid, or
- the page is not resident in physical memory (it is full).

In the first case, the memory access is invalid, and the operating system must take some action to deal with the problem. On modern operating systems, it will send a segmentation fault to the offending program. In the second case, the page is normally stored elsewhere, such as on a disk. To handle this case, the page needs to be taken from disk and put into physical memory. When physical memory is not full, this is quite simple, one simply needs to write the page into physical memory, modify the entry in the page table to say that it is present in physical memory (see the next section), write the mapping into the TLB and restart the instruction.

However, when physical memory is full, and there are no free frames available, pages in physical memory may need to be swapped with the page that needs to be written to physical memory. The page table needs to be updated to mark that the pages that were previously in physical memory are no longer so, and to mark that the page that was on disk is no longer so also (and to of course write the mapping into the TLB and restart the instruction). This process of swapping pages between physical memory and disk is known sometimes as, obviously, swapping (though the term is sometimes used to describe swapping entire processes). This process however is extremely slow in comparison to memory access via the TLB or even the page table, which lies in physical memory. Which page to swap is the subject of page replacement algorithms [4].

2.2 Universal asynchronous receiver transmitter

A UART or Universal Asynchronous Receiver-Transmitter is a piece of computer hardware that translates between parallel bits of data and serial bits. A UART is usually an integrated circuit used for serial communications over a computer or peripheral device serial port. UARTs are now built into some microcontrollers (for example, PIC16F877).

Bits have to be moved from one place to another using wires or some other medium. Over many miles, the expense of the wires becomes large. To reduce the expense of long communication links carrying several bits in parallel, data bits are sent sequentially, one after another, using a UART to convert the transmitted bits between sequential and parallel form at each end of the link. Each UART contains a shift register which is the fundamental method of conversion between serial and parallel forms.

By convention, teletype-style UARTs send a "start" bit, five to eight data bits, least-significant-bit first, an optional "parity" bit, and then a "stop" bit. The start bit is the opposite polarity of the data-line's normal state. The stop-bit is the data-line's normal state, and provides a space before the next character can start. In mechanical teletypes, the "stop" bit was often stretched to two bit times to give the mechanism more time to finish printing a character. A stretched "stop" bit also helps resynchronization. The parity bit can either make the number of bits odd, or even, or it can be omitted. Odd parity is more reliable because it assures that there will always be a data transition, and this permits many UARTs to resynchronize.

Speeds for UARTs are in bits per second (bit/s or bps), although often incorrectly called the baud rate. Standard mechanical teletype rates are 45.5, 110, and 150 bit/s. Computers have used from 110 to 230,400 bit/s. Standard speeds are 110, 300, 1200, 2400, 4800, 9600, 19,200, 28,800, 38,400, 57,600, and 115,200 bit/s.

The UART usually does not directly generate or receive the voltage levels that are put onto the wires interconnecting different equipment. An interface standard is used, which defines voltage levels and other characteristics of the interconnection. Examples of interface standards are EIA, RS 232, RS 422 and RS 485. Depending on the limits of the communication channel to which the UART is ultimately connected, communication may be "full duplex" (both send and receive at the same time) or "half duplex" (devices take turns transmitting and receiving). Beside traditional wires, the

UART is used for communication over other serial channels such as an optical fiber, infrared, wireless Bluetooth in its Serial Port Profile (SPP) and the DC-LIN for power line communication.

Today (2006), UART is commonly used with RS232 for embedded systems communications. It is useful to communicate between microcontrollers and also with PCs. Many chips provide UART functionality in silicon, and low cost chips exist to convert UART to RS232 signals (for example, Maxim MAX232) [4].

2.2.1 Synchronous

The word "asynchronous" indicates that UARTs recover character timing information from the data stream, using designated "start" and "stop" bits to indicate the framing of each character. In synchronous transmission, the clock data is recovered separately from the data stream and no start/stop bits are used. This improves the efficiency of transmission on suitable channels; more of the bits sent are data. An asynchronous transmission sends nothing over the interconnection when the transmitting device has nothing to send; but a synchronous interface must send "pad" characters to maintain synchronism between the receiver and transmitter. The usual filler is the ASCII "SYN" character. This may be done automatically by the transmitting device.

Some chips have both synchronous and asynchronous modes. These are called USARTs (for "universal synchronous asynchronous receiver-transmitters") [4].

CHAPTER 3
METHODOLOGY

Identified this project as a development project following the scheme created by the CPU designer, the CPU will solely be developed using TTL logic ICs. These logic ICs are the common digital ICs which are available at the everyday electronic stores. As the project will need hundreds of digital ICs, wire wrapping technique is viewed as the most feasible technique because it offers flexibility in construction and it can be easily reworked during debugging. Wire wrapping is also preferred because it is a fast prototyping method for circuit without the time required for designing printed circuit board.

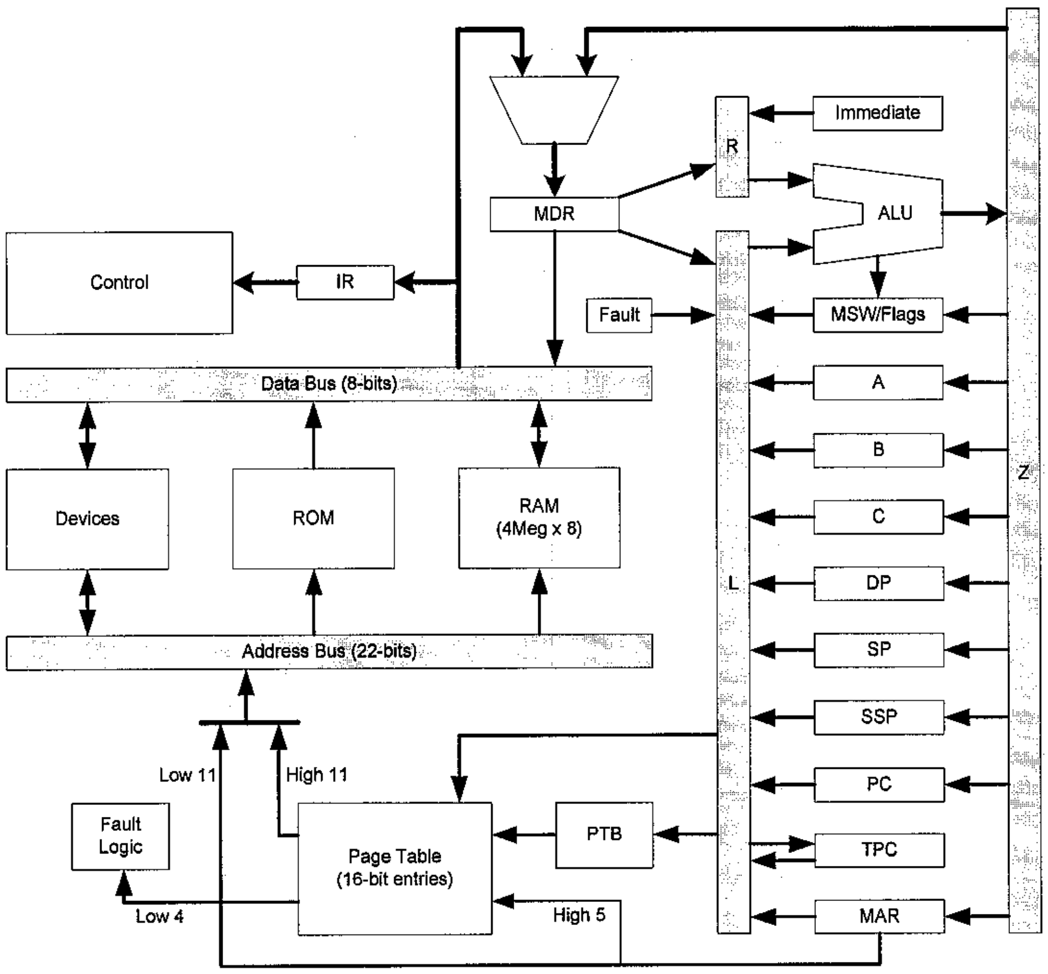
Parts of the CPU will be divided into functional parts to be mounted on several boards and later combined on a rack allowing easy access to panels and input output ports for extension. As was designed, the CPU is concatenated to a few parts installed in cards form; there are the ALU/register, control, memory, device and front panel card. These cards not only simplifies construction process but also help to ease project management as construction can be done card by card ensuring all developed cards are working before merging them together to be the CPU.

On the software side of the CPU, the assembler is needed to assemble program written for the machine and ROM burner would be needed to write PROM which stores the microcode for instruction execution. Other required hardware would be external hard disk to store larger program and a power supply to power up the CPU.

3.1 OVERVIEW

Designed machine is an 8-bit machine with the ability to run 8-bit or 16-bit arithmetic and logic operation. The 8-bit specification comes from the 8-bit data bus width. Two length of operation are supported indicates that the ALU can run two different operand word sizes.

Bit and byte order of the machine is big endian where most significant bit is numbered as zero and stored first in the memory. External interrupts and DMA is supported.



(redrawn from: www.homebrewcpu.com)

Fig. 2, Basic block diagram of the CPU

3.2 INSTRUCTION SET ARCHITECTURE

3.2.1 Operand Addressing

The machine was initially designed to be a pure one address computer. But in the design process, the operand addressing mode was slowly converted into a mixed mode with registers from accumulator was renamed to register A and other smaller details for easier compiling. So the operand addressing is not consistent throughout making this computer not an orthogonal machine.

There are nine visible register in the machine which are:

A - Accumulator. Can be addressed as 8 or 16 bits. Implied target of most operations and also used as a general load/store base register and memop operand.

B - General load/store base register, plus source operand of ALU ops and memops and target of some loads. Can also addressed as 8 or 16 bits

C - Special-purpose count register for block moves and variable shifts.

MSW - (machine status word/flags) Alu flags: **C**arry, **Z**ero, **S**ign and o**V**erflow. Control flags: **M**ode (0 for supervisor, 1 for user), **P**aging enable and **EI** (Enable Interrupts). Also, following a memory fault, a status bit, **D**ata, will appear in the saved MSW describing whether the faulting address was referencing the code or data portion of the page table.

DP - Global data pointer. Most data references are relative to a base.

SP - Stack pointer. Always pushes and pop 16 bits at a time (though doesn't need to be aligned).

SSP - Supervisor stack pointer. Used when in supervisor mode.

PC - Program Counter

PTB - Base of page table for current process in user mode. Supervisor mode base is hardwired to 0x0000. Note that the address refers to the special page table memory - not main memory.

3.2.2 Addressing mode

The available memory addressing modes are:

Register Indirect with offset - uint8(A) and uint8(B)
Frame local with offset - uint8(SP) and uint16(SP)
Global with offset - uint16(DP)
Immediate - (PC++)
Push - (--SP)
Pop - (SP++)

3.3 Microcode

The microcode is stored in five 512x8 bit ROM. The lower half will store the starting microcode while the upper half contains the continuation microcode. Since this machine is not a single cycle computer, there will be a continuation or more instructions after an initial instruction. The redirection to the next microcode index in the microcode ROM is control by the first eight bit of the microcode store. This eight bit contains exact memory location where the next instruction is positioned.

With 5 ROMs with each having a byte to contribute to the control line, there is a total of 40 control lines out from the microcode store. There are a total of 256 different instructions available as the lower half is filled with initial microcode and the ROM is 512 words in size. Full microcode listing can be found in Appendix III. The continuation microcode address of nine bits is made possible by an encoding circuit which detects the contents of NEXT field. When the NEXT field contains value but not all ones, it will become the most significant bit for the full 9-bit microcode address.

Some encoding circuit is responsible for the redirection of fetch instruction that is when the NEXT field is all ones. The circuit selects the buffering of IR (instruction register) from the DBUS into the address of the ROMs – study in further chapter.

U1	U2	U3	U4	U5
code_ptb	immval0	misc0	e_mdrlo	next0
user_ptb	immval1	misc1	l_mdrhi	next1
br_sense	e_r0	misc2	l_mdrlo	next2
l_size	e_r1	misc3	l_mar	next3
carry	e_l0	l_paging	latch0	next4
aluop0	e_l1	l_mode	latch1	next5
aluop1	e_l2	priv	latch2	next6
aluop sz	e_l3	e_mdrhi	latch3	next7

Fig. 3, Control lines (output) from the five ROMs

Outputs from the ROMs as in Figure 3 feed directly to a field decoding logic circuit decoding the outputs to discrete control lines. This is best as the registers are tri-state output registers so by encoding the controls we can keep a fairly safe bus driving scheme. As we know that no more than one driver should drive a bus. The decoding also minimizes amount of control lines as can be observed LATCH filed is a four bit output where we can select up to 16 registers to be latched.

Furthermore, some of the conditions do not occur at the same time. Such as a branch instruction does not need to do a right shift to the ALU result at the same time so does the right shift instruction. This furthermore reduces the number of control line width but with the cost of decoding circuit. Control field which adopt this concept is the MISC field which encode control signals which never occur at the same time.

0	
1	
2	halt
3	
4	trap on overflow
5	latch PTE
6	set flags (from alu op)
7	init_inst (clear
8	right shift ALU output
9	DMA acknowledge
a	latch MSW [ie] (interrupt enable)
b	do branch
c	latch MSW[in_trap]
d	commit state
e	
f	

Table 2, MISC field output decoding reducing control bit width

3.3.1 Microcode sequencer

Each microinstruction has an 8-bit "next" field, which tells which microinstruction follows.

If (next==0x00), then the next microinstruction address is the 4-bit output of a 16-line priority encoder ORed with 0x100. The least priority value is the address of the fetch microinstruction. The other values represent traps and interrupts, and the encoder value will vector control off to the appropriate interrupt or trap handling microcode. The fetch line is tied active, and so will take effect if there are no traps or interrupts pending.

If (next==0xff), then the next microinstruction address is the value of the IR (instruction register). In other words, the value of the 8-bit opcode is treated as a direct index into the microcode store.

Otherwise, the next field is ORed with 0x100 and that value is the address of the next microinstruction.

Which of the above three cases is used is determined by two control lines - MISC[INIT_INST] and a logical line which says whether next equals 0x00. INIT_INST is low active, and is asserted only during the fetch microinstruction.

Next==0x00 normally happens at the end of each sequence of microinstructions which represents an instruction. However, we also want to interrupt normal execution in the event of a trap, reset or interrupt. In the interrupt case, we want to recognize the interrupt only at instruction boundaries. That will happen normally the next time next == 0x00. For traps and reset, though, the flow needs to be broken immediately - even in the middle of a microcode instruction sequence. In these cases, there is some glue logic which will assert the asynchronous clear line of the 8-bit register holding next and resetting it to 0x00. When that happens, we in effect normalize the exceptional instruction interrupt events as if they were regular instruction boundaries. The different microcode vectors for each trap or interrupt case can then handle the cleanup for any needed state rollback or fault state collection.

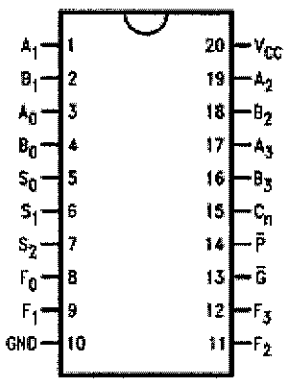
Conditional microcode branches are handled using the same mechanism as the trap's next reset scheme. If a conditional microcode branch is indicated and the condition is not met, next is reset just as it would have been had there been a trap. Care was taken when writing the microcode to ensure that no traps were possible during a microinstruction which indicated a conditional branch, so there is no ambiguity.

The conditional logic is handled by computing the various branch conditions based on the current values of the MSW condition bits. Keep in mind when looking at the logic is that when a condition is met and the machine instruction branch is taken, that we **do not** take the microinstruction branch. The branch microcode is structured so that if the branch is *not* to be taken, the microcode sequence aborts before it finished. If the branch is to be taken, the microcode continues to load the target address into PC and MAR.

CHAPTER 4
RESULTS AND DISCUSSIONS

4.1 ALU card construction and testing

Up until now, the ALU card and control card were constructed with the ALU ICs substituted with 74181s. This is due to the fact that the 74381 and 74382 ALU ICs were nowhere to be found. The substitution was done with construction of a daughter board for the ALU, this is because the size and pin configuration of 74181 and the two to be substituted are not the same.



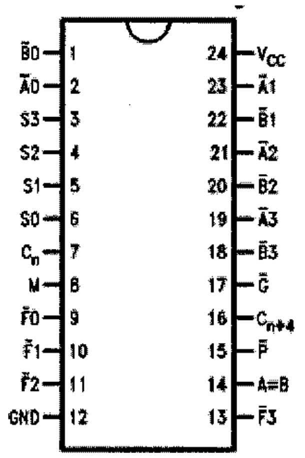


Fig. 4(a) Fig. 4(b)

(a) pinout of 74381 and (b)74181 which are not compatible physically

Another problem faced is the control lines configuration of the two different family of ALU. 74381 and 74382 have the same control line configuration as they are meant to be paired together while 74181 is a more complicated ALU which supports more functions thus having more control lines to select the function. So beside of just the ALU ICs, the daughter board will also consists of the decoding logic which will translate the function select line for 74181. Design of the decoding logic branch back to what combination logic will be selected by the control card.

It is observed that there are two control lines which select the ALU function which are ALUOP and IR. They are connected in such a way so that ALUOP is superior to IR where with proper selection of ALUOP will choose IR as the function, this is made possible with the use of 74153 4-to-1 multiplexer. The truth table for the control lines ALUOP and IR is shown in Table 1.

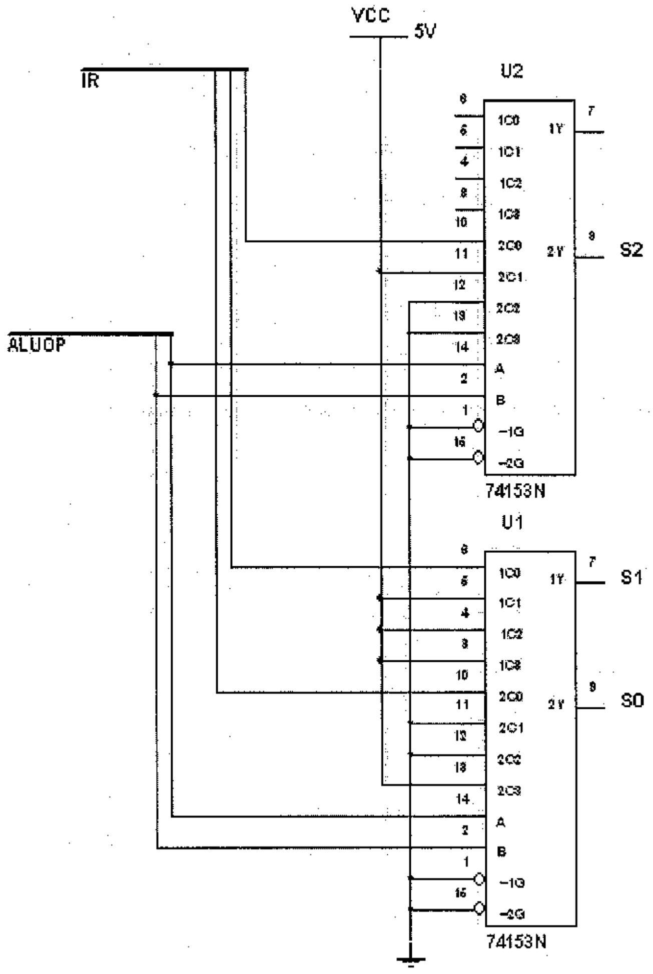
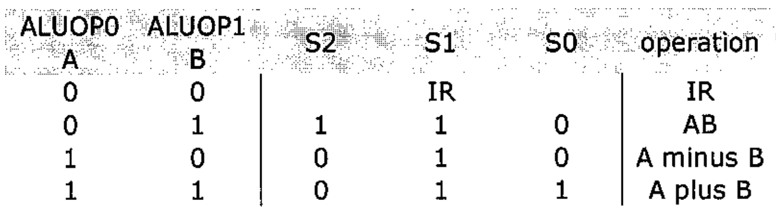


Fig. 5, multiplexer circuit to select function of the ALU from ALUOP and IR



ALUOP0 A	ALUOP1 B	S2	S1	S0	operation
0	0		IR		IR
0	1	1	1	0	AB
1	0	0	1	0	A minus B
1	1	0	1	1	A plus B

Table 3(a), ALUOP input and resulting output

IR	operation
0	
1	
2	sub
3	add
4	xor
5	or
6	and
7	

Table 3(b), corresponding operation with IR inputs

74381, 74382			74181				
S2	S1	S0	S3	S2	S1	S0	M
1	1	0	1	0	1	1	1
0	1	0	0	1	1	0	0
0	1	1	1	0	0	1	0

Table 4, truth table for decoding control lines for 74181

From the truth table (Table 3), the decoding circuit was constructed, note that only the part from ALUOP is decoded not the IR means that the circuit will have some disability compared to the original. The circuit is constructed just to gain the confidence on the construction of the circuit. Means that, the ALU will only able to perform binary addition, subtraction and AND operation for the time being. Resulting decoding circuit is shown as below after analyzing the truth table with K-map. To emulate the exact operation of 74381 and 382 will need more complex decoding circuit and is planned to be done in the future if the ICs are still nowhere to be found.

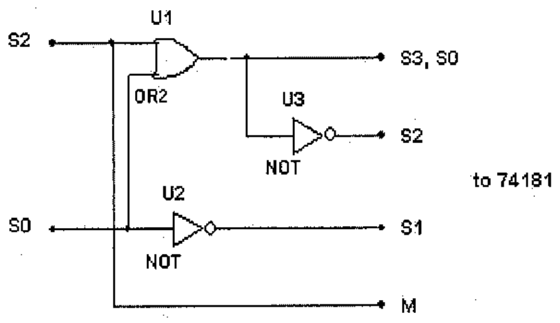


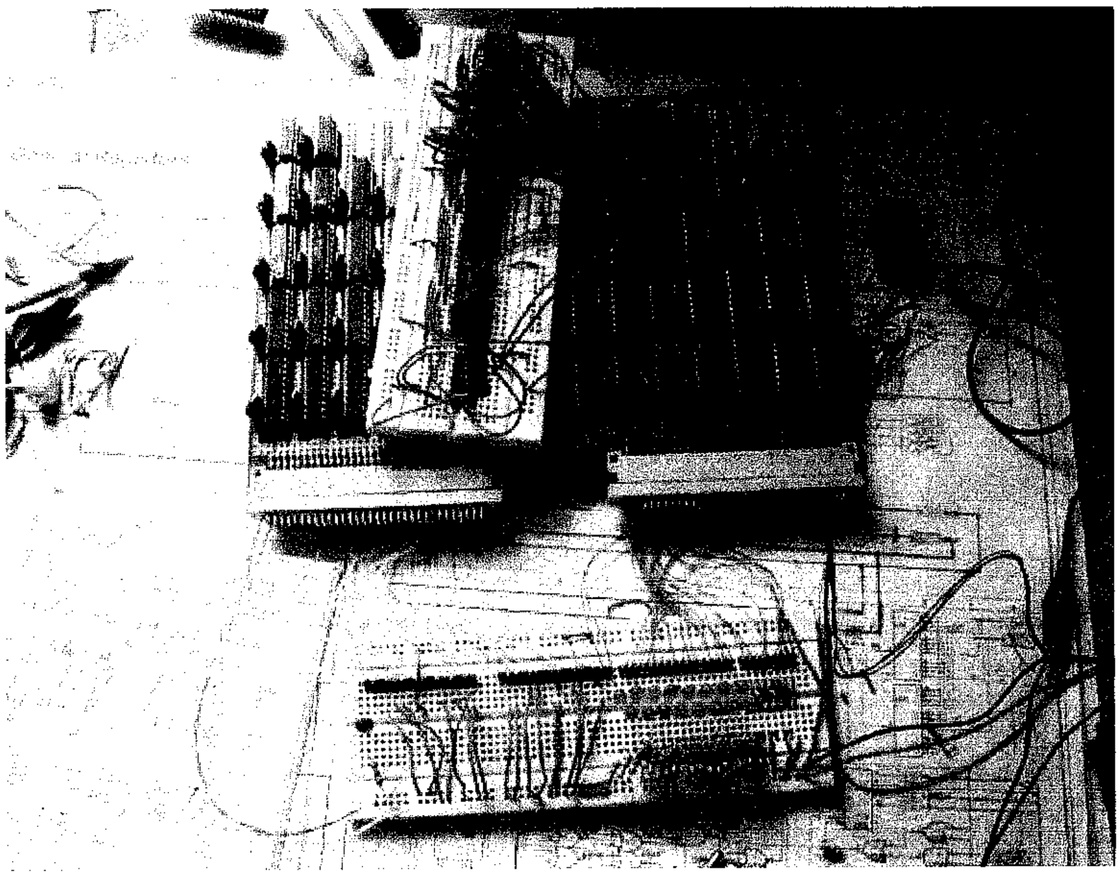
Fig. 6, designed decoding circuit for 74181

So as the substitute was designed, the ALU card can now be tested for operability. Testing for the operability requires study into the control lines, identifying buses into and out of the ALU and storage registers.

Stated requirements of the test are

- All input, output data lines and control lines are to be interfaced only through the backplane connectors R and L.
- Confirm that the ALU is working
- Confirm that registers input and output lines are working
- Check on the status flags

Flow of data is then recognized to perform the required operation based on the requirement. The requirements are set so that most of the lines are tested; these are due to the high level of uncertainty in the circuit which is the wire-wrapping technique itself, the point to point soldering of the side connectors, the designed decoder circuit for the ALU and the connections between the card and the ALU daughter board.


ALU test bed

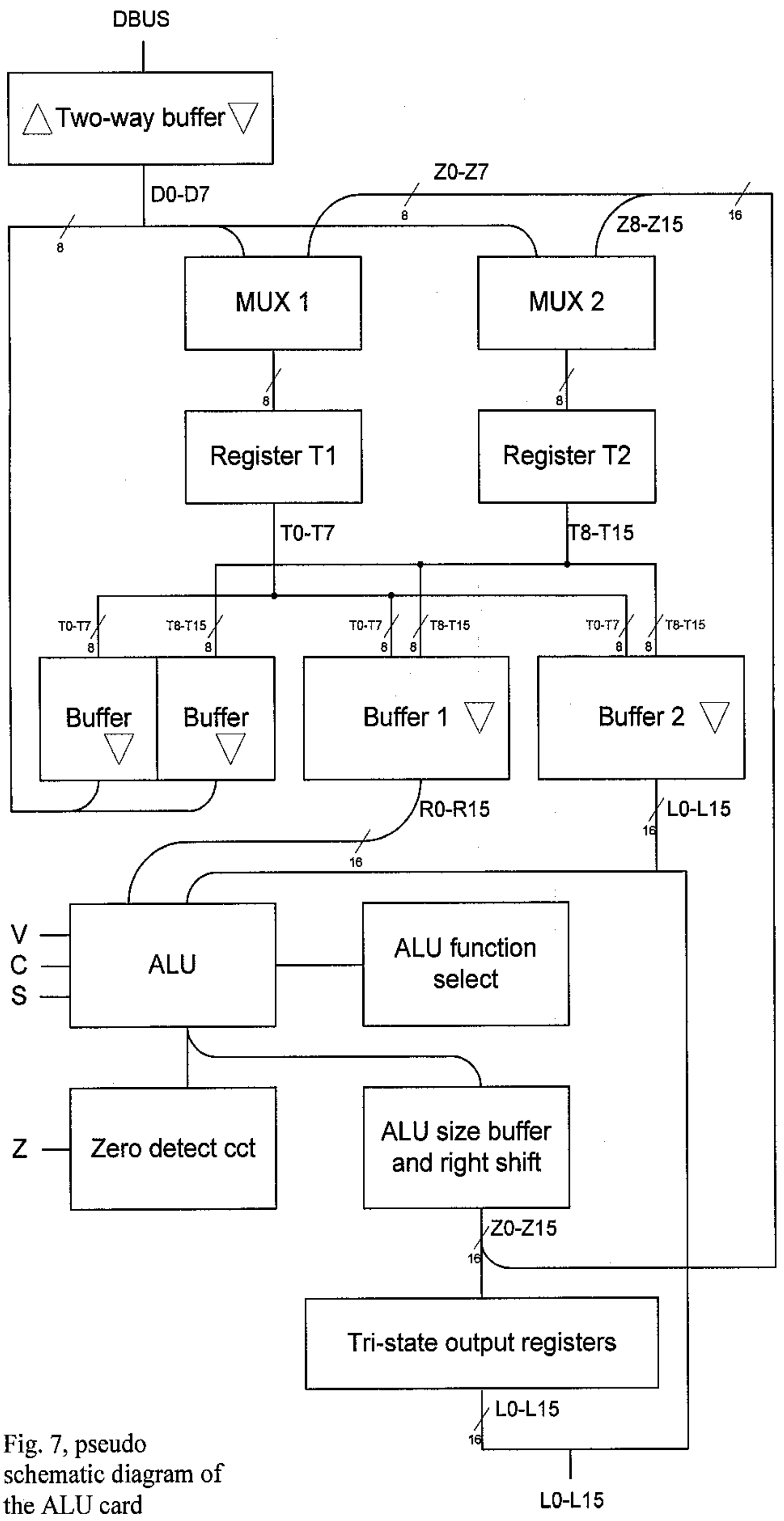


Fig. 7, pseudo schematic diagram of the ALU card

The figure presented previously (Fig. 7), shows the scope of the tested card. From the figure, there are only two busses available at the backplane to input or read data which is the L and D bus. It is noted here that each block in the schematic have at least one control line into them. Testing procedure is done by sequencing control lines from the DBUS into the ALU and storing the result in a register. The details are as follows:

1. Registers T1 and T2 are cleared (COMMIT = positive pulse)
2. Immediate value is asserted at DBUS (01100110)
3. Two-way buffer direction is selected as B to A (_RW = low)
4. The buffer is then enabled, immediate data on D bus (_DMA_ACK = high)
5. MUX 2 is set to flow D into register T2 (XL_MDR_LO = high)
6. Load register T2 with immediate data (L_MDR_LO = positive pulse)
7. MUX 1 is set to flow D into register T1 (XL_MDR_LO = low, XL_MDR_HI = high)
8. Load register T1 with immediate data (L_MDR_HI = positive pulse)
9. Buffer 1 and Buffer 2 are set to assert both bus R and L with the same content of bus T as right and left operand into the ALU – 0110011001100110 (_ER_MDR = low, _EL_MDR = low), content of bus L which is already connected to LEDs can be viewed
10. ALU operation is set to ADD (ALUOP0 = high, ALUOP1 = high)
11. Use of carry is prohibited (USE_CARRY = low), but in design this line is read as active low by the substitute ICs, **there is a carry in.**
12. ALU operation size selected as 16 bits (ALUOP_SZ = low)
13. Result is not shifted right by one bit (_DO_RSHIFT = high), result of the bitwise addition of the same operands with a carry in is now on the Z bus
14. Result is then stored into one of the registers, selectively register C (L_C = positive pulse, clock in)
15. To read the content of register C, first disable the buffering of operand into bus L by Buffer 2 (_EL_MDR = high)
16. Read the content of register C through bus L (_EL_C = low), result of the addition can now be viewed through the LEDs
17. Reading the flags – only zero flag is connected to the LED in the test (_SET_FLAGS = low, L_MSW = positive pulse)

4.1.1 Results

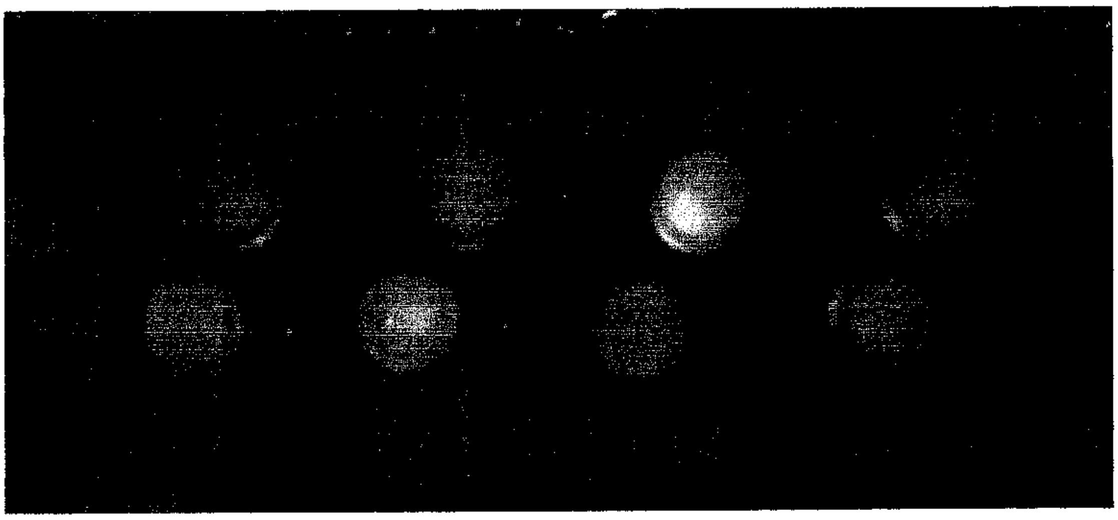


Fig. 8, LEDs showing content of L bus

Figure above shows the content of L bus containing 0110011001100110 buffered from register T1 and T2 which is also the addend and augend of the ALU. Binary addition is then done with a carry in.

$$\begin{array}{cccccccccccccccccc}
 & 1 & 1 & & & 1 & 1 & & & 1 & 1 & & & 1 & 1 & & 1 \\
 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
+ & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
\hline
 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1
\end{array}$$

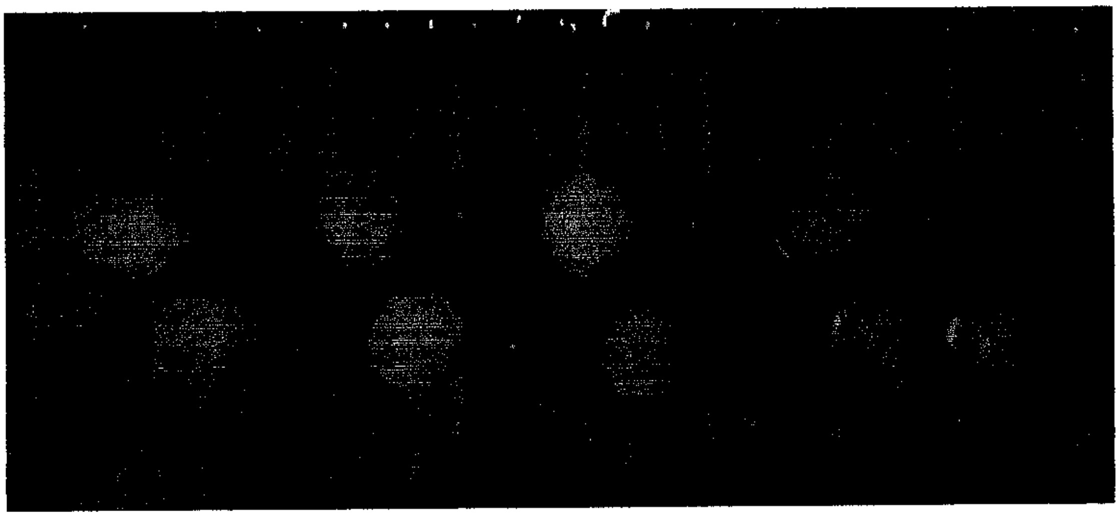


Fig. 9, LEDs on L bus showing result of addition from register C

Reading from register C where the result is stored confirms the addition operation concluding the decoding circuit, ALU connections and busses D, T, R, L and Z are working fine.

4.2 Control card study

On reset, registers U6 – U10 (74273s, Figure 11) that temporarily store microcode are cleared. NEXT filed is all zeros and circuit that detects NEXT filed content for all ones asserts a high value (active low output) which forces the multiplexer input B to be high. Note that B also signifies the ninth bit into the microcode storage to address the top half of the PROM. Another glue logic checks for NEXT field for all zeros and forces the multiplexer input A to low.

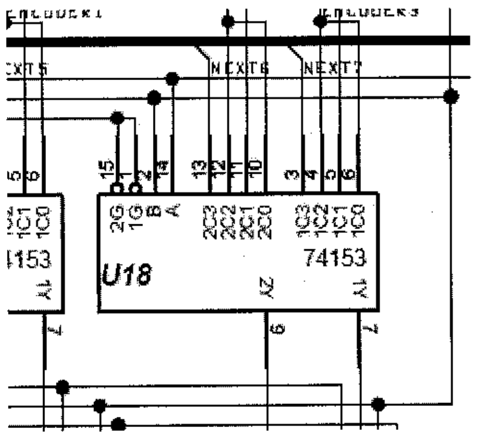


Fig. 10, Instruction multiplexer from the schematic

B (also the ninth bit to PROM)	A	Microcode source
0	0	Fault
0	1	IR (D bus)
1	0	Fault
1	1	NEXT

Table 5, Instruction multiplexing table

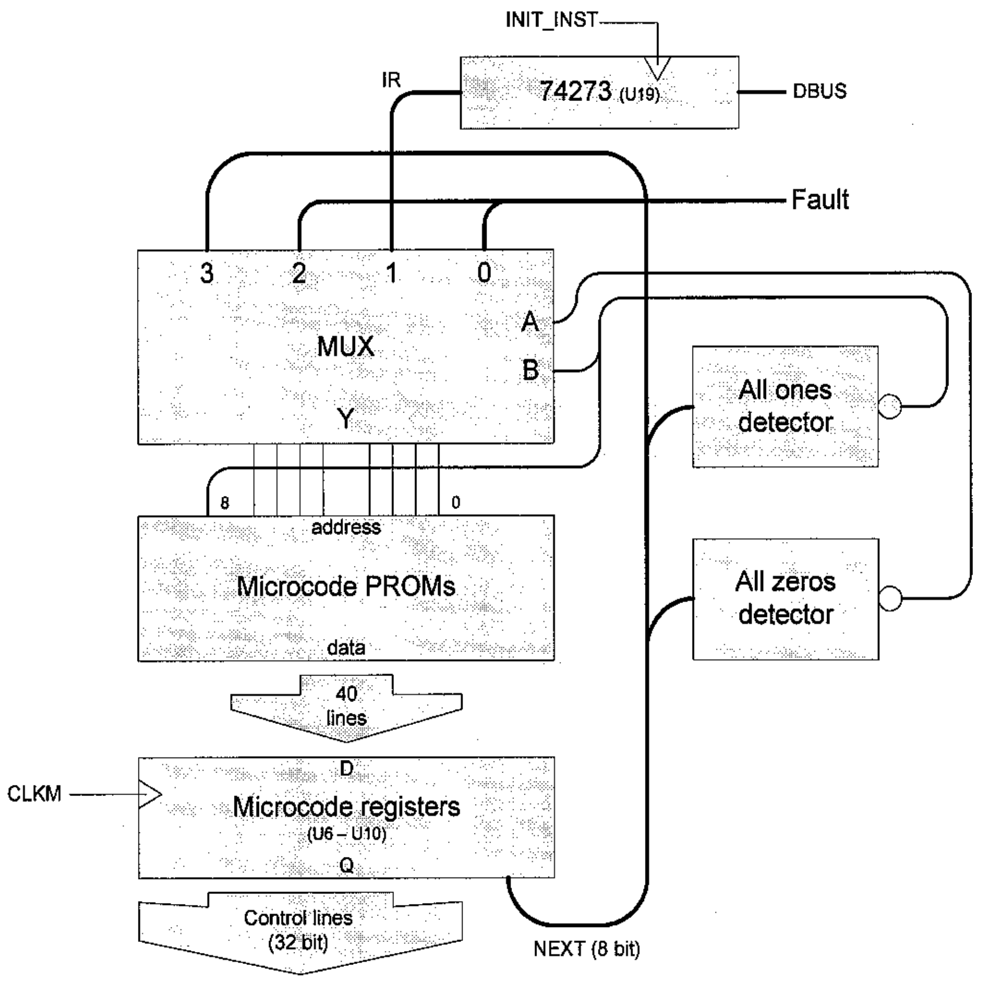


Fig. 11, Simplified block diagram of microcode sequencer circuit

With BA = 10, fault circuit is selected as the instruction select. Since the reset button also clears all faults (0x00 from fault circuit, please refer to the appendix for faults and interrupts schematic), no fault is pending and fetch instruction (0x100) is now driven into the microcode PROMs address after the rising edge of clock (CLKS). On falling edge of clock, FETCH instruction from the microcode PROM is now clocked into the temporary registers U6-U10. NEXT field is now all high, 0xff causing A = high, and thus putting B to low.

With BA = 01, the multiplexers select IR as instruction select. Right now all control lines corresponding the instruction has been relayed to the circuit around the control card. For the fetch instruction, one example microinstruction is incrementing the MAR. On rising edge of clock, instruction on D bus is clocked into a register (U19 by INIT_INST line) due to FETCH instruction. And instruction from D bus now is driven into the microcode store. Upon falling edge of the clock, selected instruction from the microcode PROM is now clocked into the temporary registers U6-U10. NEXT field now would be dependent on the instruction selected.

One clock instruction like NOP (no instruction) would have all zeros NEXT field which forces A becoming low from the _NEXT0 glue logic. And B equals high. We had already met this condition before BA = 10. All zeros NEXT field marks the end of an instruction and where faults and interrupts are checked and served. If there are no fault and interrupt pending, FETCH is selected.

For multiple clock cycle instructions, NEXT field would be some value between 0x01 to 0xfe as 0x00 marks instruction boundary and 0xff only called by FETCH instruction. So NEXT field is neither all zeros nor all ones, circuit detecting both conditions have active low outputs so both B and A will be high causing NEXT driven into microcode store (refer to table 5).

The process repeats itself as each instruction is executed. Clocks coordinate devices clocking data onto busses and execution of instruction.

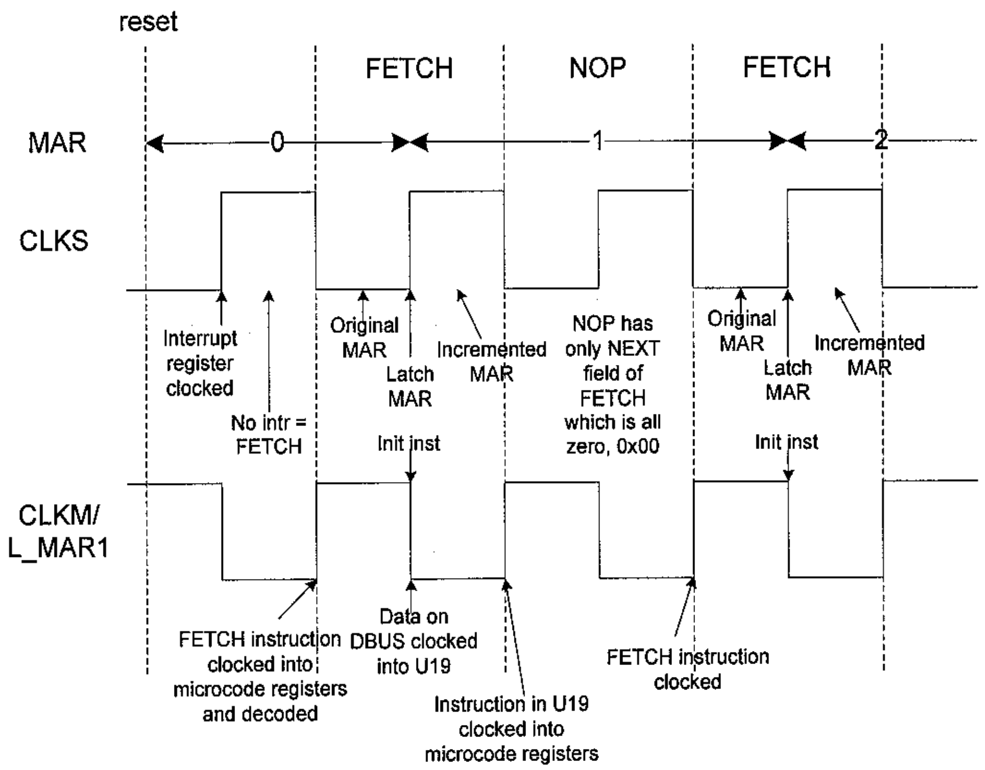

reset
FETCH
NOP
FETCH
MAR
0
1
2
CLKS
Interrupt register clocked
No intr = FETCH
Original MAR
Latch MAR
Incremented MAR
Init inst
NOP has only NEXT field of FETCH which is all zero, 0x00
Original MAR
Latch MAR
Incremented MAR
Init inst
CLKM/ L_MAR1
FETCH instruction clocked into microcode registers and decoded
Data on DBUS clocked into U19
Instruction in U19 clocked into microcode registers
FETCH instruction clocked

Fig. 12, Instructions fetching and execution

4.3 Clocks

During the build-up, clocks were studied to further understand how the computer works. Without clocks, the connected logic will be dead, clocks is needed basically to change content of registers or flip-flops. With sequentially changing content of registers, data can be passed through the digital circuits. In this computer, there are basically two clocks which are CLKS and IOCLK. CLKS and its complement CLKM are connected to most parts on the control card while the IOCLK clocks the device and memory card.

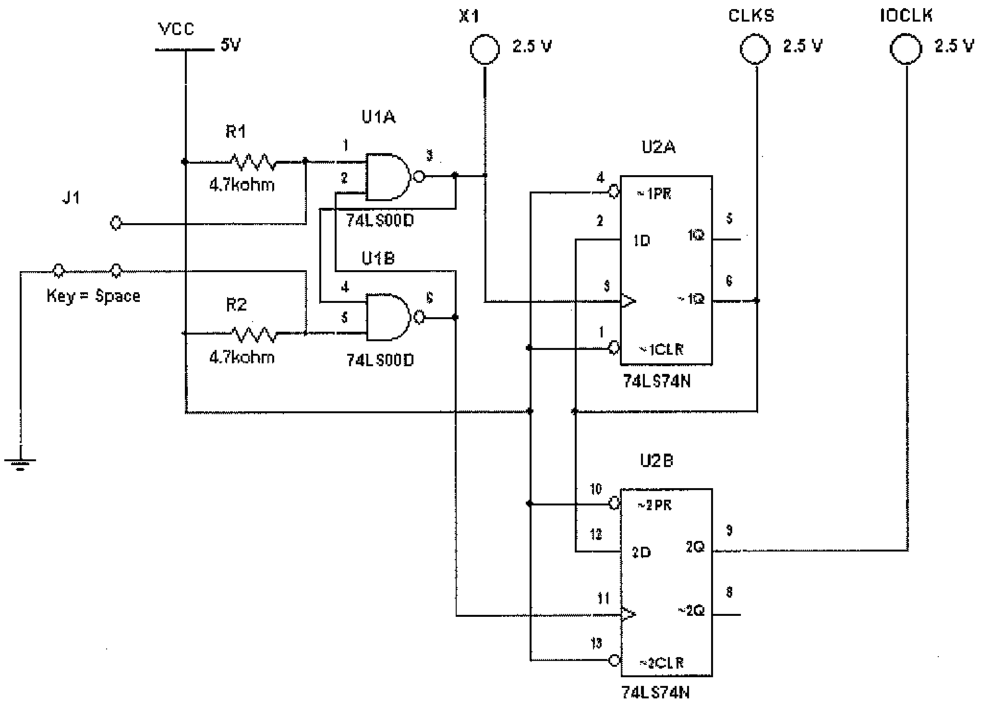


Fig. 13, Simplified and simulated schematic of the clock generator

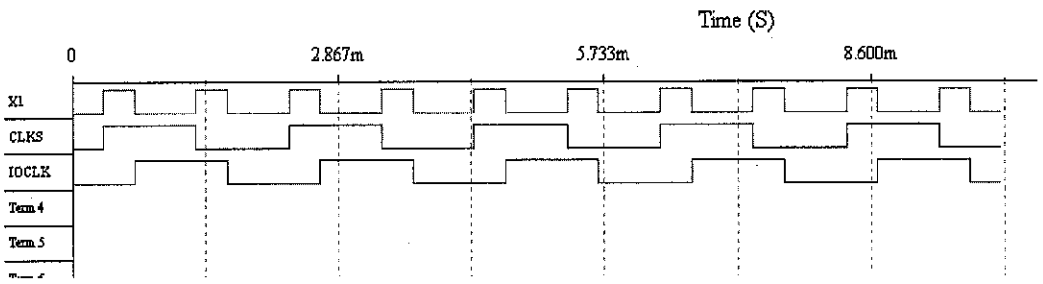


Fig. 14, Clocks waveform with respect to input clock X1

The skewed clock design is important to make sure proper clocking of data from the devices to other peripherals. Best example would be the enabling of output of the boot PROM by IOCLK which is active low and data on the D bus is then clocked into the instruction register at the microcode store circuit by CLKS rising clock edge.

4.4 Microcode sequencing example

In this example, we will simplify the machine and focus on the basic and most critical part. So we will never turn paging to ON which will be cleared during reset, this will map A (address) bus to be the same as MAR, this is done by the memory card (page table in Appendix II). At the device side, we will only consider mapping ranges for the boot ROM which ranges from 0x00 until 0x3fff. With paging off, we must make sure that the MAR value does not exceed 0x3fff or another device will be selected. Following these assumptions, a simplified block diagram is drawn.

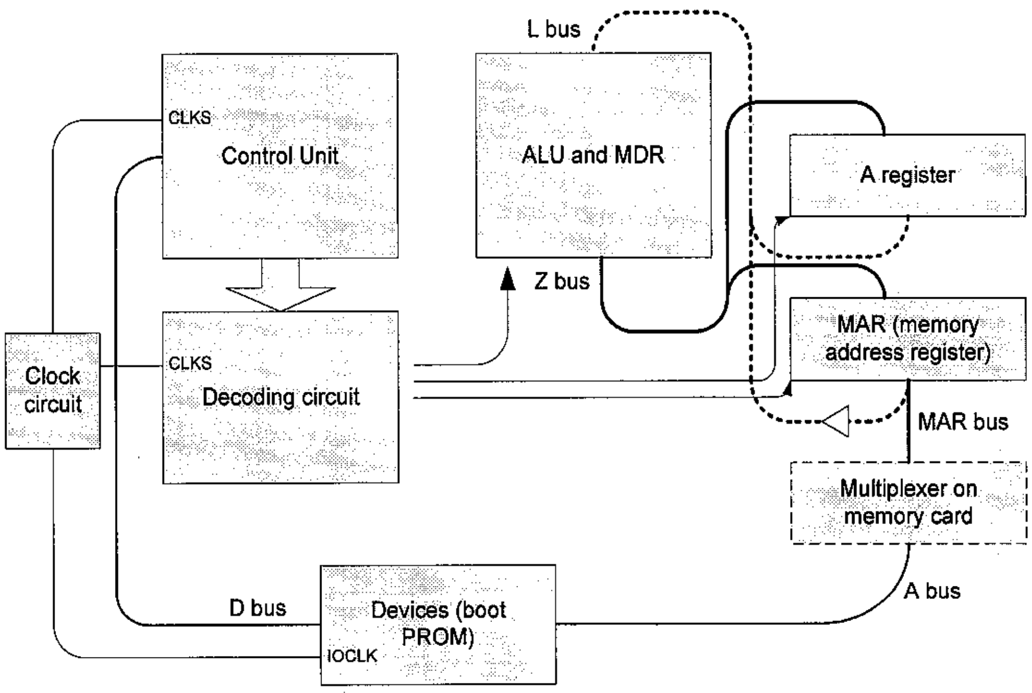


Fig. 15, Simplified CPU for the example

Now we consider a very simple program in assembly as follows which assumed to be burnt into the boot PROM.

```
Add   Inst
00    35    |    add.8     A,#1
01    92    |    copy      B,A
```

This program is a two instruction program but it does not indicate that it can be completed in two clock cycles. As we already know, this is a complex instruction machine and not all instructions are done in a clock cycle. Now we will look into the microcode for these instructions extracted from Appendix III.

```
0x35 add.8 A,#1 ; INC_TO_Z(R_MDR),L(R_MDR,LWORD),NEXT(Aluop8)
```

The add.8 instruction is not a one clock cycle instruction because it calls for another microinstruction in the next field which is Aluop8.

```
0x112 Aluop8 ;
E_L(R_A),E_R(ER_MDR),ALU(OP_IR13,BYTE,NO_CARRY),L(R_A,LBYTE),MISC(M_S
ET_FLAGS),NEXT(Fetch)
```

Aluop8 is the microinstruction needed when doing an 8-bit ALU operation. Means it is not only called by add.8 but also other 8-bit operations. Only here, the next field is fetch which marks the end of the instruction. From here we say that add.8 is a two clock cycles operation as it requires two microinstructions to complete the instruction.

```
0x92 copy B,A ; TO_Z(R_A),L(R_B,LWORD),NEXT(Fetch)
```

Copy B,A on the other hand is a one clock cycle instruction because it does not require another microinstruction to finish the instruction, instead it directly calls the fetch instruction

As discussed in previous section, at reset, MAR will be cleared yielding all zeros. Memory card generates the address using the value of MAR produces all zeros for the address bus too. A little detail were left off in figure 12, actually the address registers at the memory card are clocked a little later than the MAR which is by CLKM. Meaning that the A bus and MAR bus content are not the same in time base, see Figure 16. Anyway, these all zeros memory will map to the boot room and the corresponding content of the address will now relayed onto the D bus.

The executed FETCH instruction will latch the content of D bus at A = 0x00, which is 0x35. 0x35 refers to the location of microcode in the PROMs. The microcode is retrieved and stored into the microcode registers (U6 – U10). Later, the instruction was decoded to perform the first wave of microinstruction. With the NEXT field containing 0x12, which is neither all ones nor all zeros making the instruction multiplexer selecting NEXT as the instruction (Figure 11 & Table 5).

NEXT from the microcode register is buffered directly into the microcode store (PROM) with the additional bit from B, we have a complete nine bit instruction address of 0x112. With the same convention as earlier, instruction is clocked into the microcode registers, instruction decoded and initialized. The NEXT field is now being considered, with all zeros (0x00) marked the end of the instruction; fetch will be selected as the next instruction. Fetch will basically increment the MAR so that the next instruction can be executed. Address bus is clocked half a clock period after MAR being incremented to ensure proper latching of instruction. MAR already at 0x02, address bus is still at 0x01 and instruction copy B,A now latched into U19.

Instruction done, the next field of copy B,A is fetch where MAR is incremented and the microcode sequencer looks for further instructions.

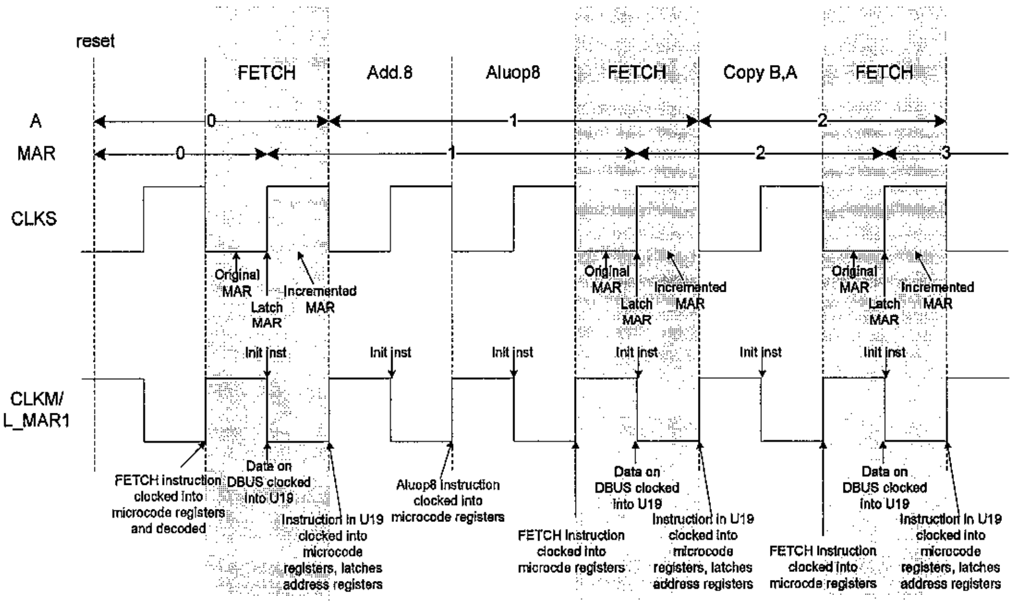

reset
FETCH
Add.8
Aluop8
FETCH
Copy B,A
FETCH
A
MAR
0
1
2
3
CLKS
Original MAR
Latch MAR
Incremented MAR
Init inst
CLKM/ L_MAR1
FETCH instruction clocked into microcode registers and decoded
Data on DBUS clocked into U19
Instruction in U19 clocked into microcode registers, latches address registers
Aluop8 instruction clocked into microcode registers
FETCH instruction clocked into microcde registers

Fig. 16, Timing diagram for the example

4.5 Construction Diary

There was not much problem faced during testing of the front panel except for that some parts were not available and the high frequency crystal clock did not work the first time. Also a multiplexer IC (74157) was found broken causing the clock not redirected. Chips which are not available even from the LS family are

74F533 (Octal Transparent Latch with 3-STATE Outputs)
74F534 (Octal D-Type Flip-Flop with 3-STATE Outputs)

Substitutes were designed for the two ICs, realizing that 74533 is the same to 74373 except with inverted outputs and 74534 is the same to 74374 also with inverted output. 74373 and 74374 are both available and the substitutes were designed by pairing each with an inverted buffer (74240) at the output. Other functionality of the front panel card was also tested like the stop clock, variable clock, manual clock, reset switch and other switches and LED drivers.

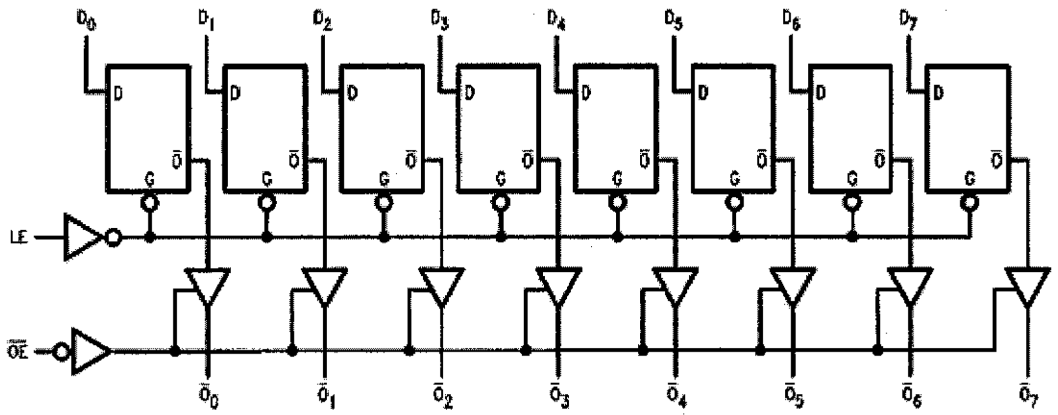


Fig. 17, 74533 logic diagram (from Fairchild semi, 74F533 datasheet)

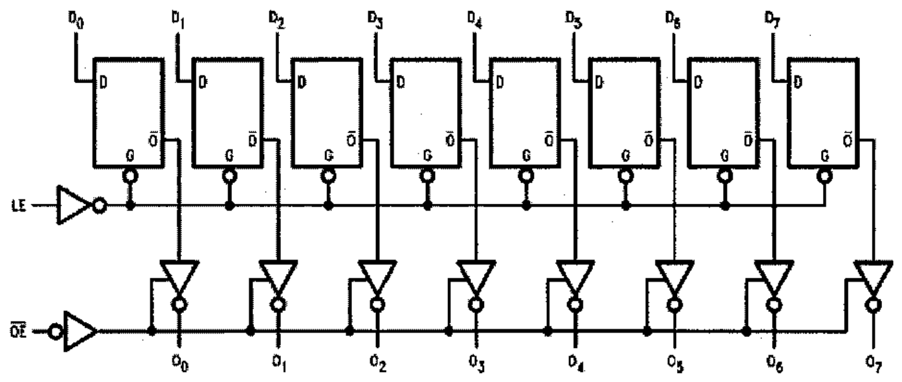


Fig. 18, 74373 logic diagram (from Fairchild semi, 74F373 datasheet)

The first time four of the cards – except the memory card – were installed in the cage and powered up, there was no sign of life at all except for the clock. When the address line is at 0x00h which is after reset, device selected to drive the D bus is the boot PROM. The address lines will not be all zeros without the memory card, so the address lines were hardwired to ground which means the boot PROM will always be selected independent from the value of MAR. This modification is necessary as there is not enough space in the cage to accommodate all the cards due to the ALU substitute breadboard on top of the ALU card.

4.6 Fibonacci counter test

The CPU can now works with switched-in instruction because there is no program has been written for it. In order for a program to run, the address lines would need to sequence itself and basically increments with each instruction execution. This is not possible until now because the memory card is not installed and it is responsible of generating the address from MAR - the MAR (memory address register) already increments itself after each instruction.

To further test the CPU, the ALU substitute board was changed to a lower profile version allowing all cards to be in the cage. A simple program was arranged in machine language to show that the machine is actually working. Written program is a Fibonacci counter that counts up the Fibonacci series. Due to hardware limitation (using registers to store result, which is 16-bit wide), the CPU will only be able to count up to 65,535 (2^16-1)

Fibonacci series [5] :

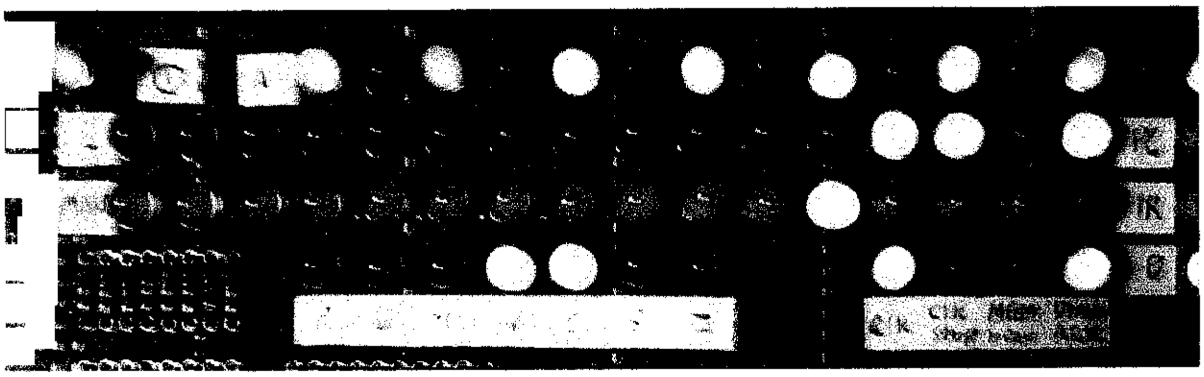
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, …

The program:

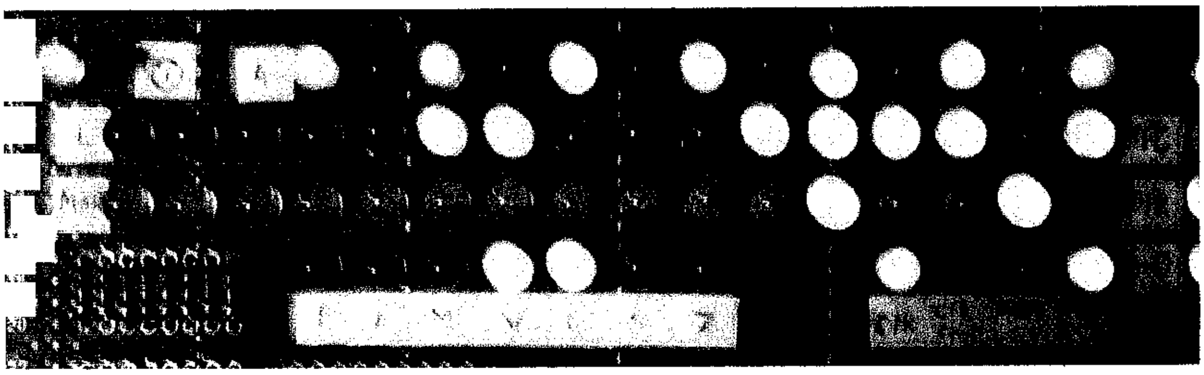
```
Add     Instr      Mnemonic            Description

00      35     add.8 A,#1
01      46     xor.16      A,A   A=0
02      35     add.8 A,#1
03      52     sex   A     A=1
04      3e     add.16      A,A   A=2
05      3e     add.16      A,A   A=4
06      3e     add.16      A,A   A=8
07      3e     add.16      A,A   A=16
08      cb     copy  SP,A  SP=16, initialize loop address

09      2a     nop0

0a      35     add.8 A,#1
0b      46     xor.16      A,A   A=0
0c      92     copy  B,A   B=0
0d      35     add.8 A,#1
0e      52     sex   A     A=1
0f      b6     copy  DP,A  DP=1

10      2a     nop0

******************************Fibonacci loop*****
11      ba     copy  A,DP
12      96     copy  C,A
13      3f     add.16      A,B
14      b6     copy  DP,A
15      f2     copy  A,C
16      92     copy  B,A
17      ba     copy  A,DP

*********************************************

18      f6     copy  A,SP  A=16
19      32     br    A     MAR=16, copy A into MAR
```

The Instr column above is burnt into an EPROM as the data corresponding to Add as address.

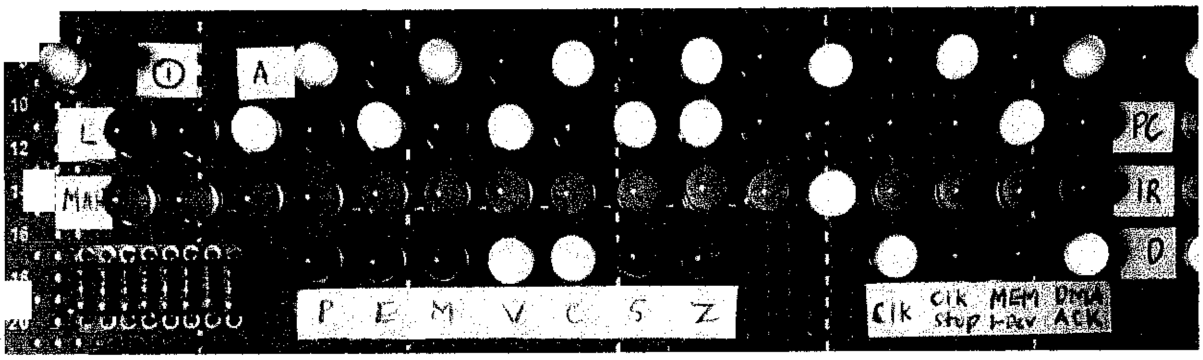
Some of the results:



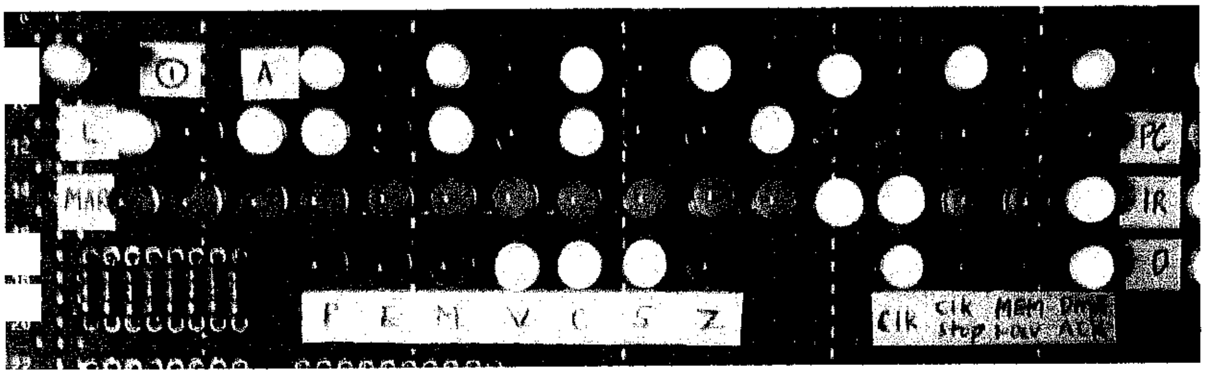
L = 2^3 + 2^2 + 2^0 = 13



L = 2^10 + 2^9 + 2^5 + 2^4 + 2^3 +2^2 +2^0 = 1597



L = 2^13 + 2^11 + 2^9 + 2^7 + 2^6 + 2^1 = 10946



L = 2^15 + 2^13 + 2^12 + 2^10 + 2^8 + 2^5 = 46368

4.7 Case and front panel construction

The CPU casing design started immediately after the successful loading of Fibonacci test. The casing was designed to house the card cage, power supply, front panel and other peripherals. Connectors available from the backside of the case are the power cord and the two serial ports. As for the material, it was build with the same material used for the card cage.

Giving a platform already used to, design commence smoothly but with added complexity to enhance ergonomics. The front panel side was designed to slant aiding visibility for the user of the front panel. Casing of the CPU is covered with acrylic letting observers to view the inside components.

4.8 Machine validation suite

Missing parts mailed by Bill Buzbee arrived just in time (toward finishing touches of the front panel). So the hard-to-find parts are now available like the 74381 and 74382 (ALU chips), the substitute ALU board is no longer needed now. Other parts included in the package are SRAM chips for the memory, page table entry, and device space, UART chips, real time clock chip and HP hex displays.

Proposed next step is to run the validation suite which is a series of tests. The tests range from basic like instruction tests to advanced tests like branching and memory tests (more details at www.homebrewcpu.com\validation_suite.htm). All these tests are written in Magic-1 assembly language, tests result and its description is summarized in the following table. Then the simple memory test will be discussed.

Test name	Tested operation	Result
B 001	`Load immediate - 8 bit, no sign extend nops`	Pass
B 002	`16-bit load immediate`	Pass
B 003	`16-bit load immediate using signed extended 8-bit immediate`	Pass
C 001	`register to register copy`	Pass
C 002	`lea all 16-bit offsets using SP, A, B, DP and PC.  Targets both A and B`	Pass
D 001	`8-bit memory loads`	Pass

D 002	`16-bit memory loads`	Pass
D 003	`ldclr`	Pass
E 001	`8-bit memory stores`	Pass
E 002	`16-bit memory stores`	Pass
E 003	`memcopy`	Pass
E 004	`memcopy`	Pass
F 001	`push/pop`	Pass
G 001	`add.8`	Pass
G 002	`add.16`	Pass

G 003	`adc a,a`
	`adc a,b`
		Pass

H 001	`sub.8`	Pass
H 002	`sub.16`	Pass
H 003	`sbc a,b`	Pass
I 001	`and.8`	Pass
I 002	`and.16`	Pass
J 001	`or.8`	Pass
J 002	`or.16`	Pass

J 003	`xor.16    a,a`
	`xor.16    a,b`
		Pass

L 001	`vvshl.16, vshr.16, sex`	Pass
L 002	`shl.16, shr.16`	Pass
M 001	`cmp.8`	Pass
M 002	`cmp.16`	Pass
O 001	`call, return, enter 8, enter 16`	Pass

P 001	`bset.8 a,mask,d8`
	`bclr.8 b.mask,d8`
		Pass

P 002	`bset.16 a,mask,d8`
	`bclr.16 b.mask,d8`
		Pass

Q 001	`br`
	`br.eq`
	`br.ne`
	`br.lt`
	`br.ge`
	`br.gt`
	`br.le`
		Pass

Q 002	`br.ltu`
	`br.geu`
	`br.gtu`
	`br.leu`
		Pass

R 001	`cmpb.eq.8 a,u16(dp),e8`
	`cmpb.eq.8 a,u8(sp),d8`
	`cmpb.eq.8 a,u8(b),d8`
	`cmpb.eq.8 a,i8`
	`cmpb.eq.8 a,0`
	`cmpb.eq.8 a,b`
		Pass

R 002	`cmpb.ne.8 a,u16(dp),e8`
	`cmpb.ne.8 a,u8(sp),d8`
	`cmpb.ne.8 a,u8(b),d8`
	`cmpb.ne.8 a,i8`
	`cmpb.ne.8 a,0`
	`cmpb.ne.8 a,b`
		Pass

R 003	`cmpb.eq.16 a,u16(dp),e8`
	`cmpb.eq.16 a,u8(sp),d8`
	`cmpb.eq.16 a,u8(b),d8`
	`cmpb.eq.16 a,i16`
	`cmpb.eq.16 a,extii8`
	`cmpb.eq.16 a,0`
	`cmpb.eq.16 a,b`
		Pass

R 004	`cmpb.ne.16 a,u16(dp),e8`
	`cmpb.ne.16 a,u8(sp),d8`
	`cmpb.ne.16 a,u8(b),d8`
	`cmpb.ne.16 a,i16`
	`cmpb.ne.16 a,extii8`
	`cmpb.ne.16 a,0`
	`cmpb.ne.16 a,b`
		Pass

S 001	`cmpb.lt.8 a,u16(dp),e8`
	`cmpb.lt.8 a,u8(sp),d8`
	`cmpb.lt.8 a,u8(b),d8`
	`cmpb.lt.8 a,i8`
	`cmpb.lt.8 a,0`
	`cmpb.lt.8 a,b`
		Pass

S 002	`cmpb.le.8 a,u16(dp),e8`
	`cmpb.le.8 a,u8(sp),d8`
	`cmpb.le.8 a,u8(b),d8`
	`cmpb.le.8 a,i8`
	`cmpb.le.8 a,0`
	`cmpb.le.8 a,b`
		Pass

S 003	`cmpb.lt.16 a,u16(dp),e8`
	`cmpb.lt.16 a,u8(sp),d8`
	`cmpb.lt.16 a,u8(b),d8`
	`cmpb.lt.16 a,i16`
	`cmpb.lt.16 a,extii8`
	`cmpb.lt.16 a,0`
	`cmpb.lt.16 a,b`
		Pass

S 004	`cmpb.le.16 a,u16(dp),e8`
	`cmpb.le.16 a,u8(sp),d8`
	`cmpb.le.16 a,u8(b),d8`
	`cmpb.le.16 a,i16`
	`cmpb.le.16 a,extii8`
	`cmpb.le.16 a,0`
	`cmpb.le.16 a,b`
		Pass

U 001	set up page table identical to no paging, turn paging on, do a few simple ops, turn paging off do a few simple ops.	Pass
U 002	from_sys	Pass
U 005	ldcode.8, stcode.8	Pass
W 001	sram addressing	Pass
Y 002	shladd	Pass
Pageon	All test with paging on	Pass
Usermode	All test	Pass
Memtest	Simple memory test	Pass

Table 6, validation suite summary

```
start:
    ld.16       a,0
    copy        ptb,a        ; set page table base to supervisor
    ld.16       b,0x0000     ; address of low 2K bytes of device rom
    ld.16       a,0x8000 + 0x4000   ; flags to set page present,
                                      writeable & device space
    wcpte       a,(b)
    wdpte       a,(b)        ; set up paging for first 2K of rom (i.e. -
                               this code that is running now
    ld.16       b,0x9000     ; pick a virtual address to try
    ld.16       a,0x8000 + 0x4000 + 0x2000 + 2    ; set to present,
                               writeable, SRAM page #2 [3rd 2K page]
    wdpte       a,(b)
    copy        a,msw
    or.16       a,0x80
    copy        msw,a        ; turn paging on
    ld.16       a,0x5335     ; get a word of data
    ld.16       b,0x9000     ; point to newly mapped space
    st.16       0(b),a       ; store - should put 0x53 in 0x9000 and
```

```
                                     0x35 in 0x9001, these in turn should map
                                     to physical SRAM address 4096 and 4097
                                     [0x1000 & 0x1001]
    ld.16           a,0            ; clear out A
    ld.8            a,0(b)
    cmpb.eq.8 a,0x53,good1
    halt                           ; FAILURE if halted here
good1:
    ld.8            a,1(b)
    cmpb.eq.8 a,0x35,good2
    halt                           ; FAILURE if halted here
good2:
    nop0
    nop0
    nop0                           ; put some distance between fail and pass
                                     to more easily distinguish them
    halt                           ; PASS if halted here
```

Above is the actual lines in 'memtest.s' which test the memory by storing data in a location and later verifying it by reading back the location and comparing it with the expected value. Pass or fail is observed by looking at the address at which the program halts, these addresses are listed in the ".lst" file. In contrary, we can also check the content of that address (0x1000 and 0x1001) manually through the front panel switches which should show 0x53 and 0x35 respectively, otherwise the CPU failed the memory test.

4.9 Serial port and terminal interfacing

Passing all tests in the validation suite, confidence level on the operability of the machine now had increased. Next step is to run the loader program and interface the CPU to a computer so to view the output and provide a means to load programs into the CPU.

UART (universal asynchronous receiver/transmitter) chip is the functional unit which does serial communication with another computer. It is a monolithic IC which is dedicated to do just that, but before we are able to use it, it had to be initialized first setting up the baud rate, parity bit and etc. As it is a specialized chip, it would be a sure go given a correct electrical connections and terminal settings.

For the CPU, the serial ports are not only used to issue command to and from the CPU but also used to transfer program images. For the latter, a high speed connection is always desired so to reduce waiting time. And for a high speed connection to occur, the computer had to have the ability to control the flow of information being transferred. High a mount of data is allowed only if the computer is able to handle it and this communication between two computers requires handshaking. Serial connection between two computers require the use of null cable modem where data transmit line from a header is connected to data receive on the other header. Schematic of the null cable modem with full handshaking used for the communication is pictured below.

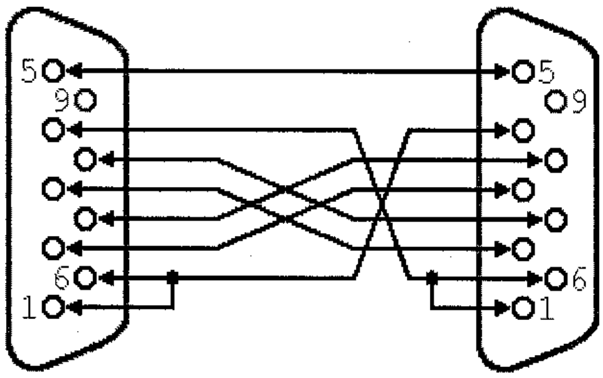
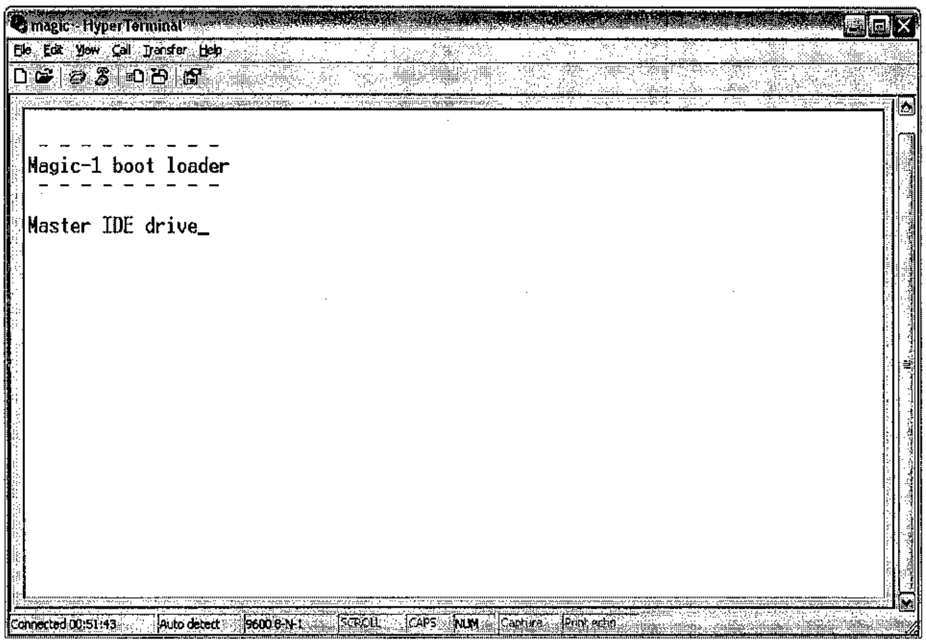


Fig. 19, Full-handshaking null modem cable connection [6]

Successful communication was established after building the correct null cable modem. A screenshot of the terminal while the CPU boots up is as follows



```
 - - - - - - - - - -
Magic-1 boot loader
 - - - - - - - - - -

Master IDE drive_
```

Fig. 20, Screenshot of the terminal

4.10 Performance

Since the machine was not that complete to run the benchmarking program, result from Bill Buzbee's benchmarking is included instead. Take note that the comparison was done to the some of the old machine back in the days, as the CPU is too slow if compared to computer nowadays.

Magic-1's new Dhrystone benchmark score is 506. To put that in perspective, here are some historical numbers:

Machine	CPU	OS	Score
Apple IIe	65C02 - 1.02 Mhz	DOS 3.3	37
- CPM -	Z80 - 2.5 Mhz	CPM-80 v2.2	91
IBM PC/XT	8088-4.77 Mhz	Coherent	275
PDP-11/34A	w/FP-11C	Unix V7m	449
Magic-1	**Magic-1 - 4.0 Mhz**	**M-1 Homebrew OS**	**506**
Macintosh 512	68000 - 7.7 Mhz	Mac ROM O/S	625
IBM PC/AT	80286 - 8 Mhz	Venix/286	1254
VAX 11/780	- -	Unix 4.2BSD	1441

Table 7, Magic-1 benchmark

As we can see, the original Magic-1 runs at 4MHz. It is in doubt that the clone can run at 4MHz, this is because some parts are not fast enough compared to what Bill's used on his machine.

CHAPTER 5
CONCLUSION AND FUTURE WORK

Chapter one of this report mentioned that current computer systems are complex and it would be a challenge to show how all the different components are assembled to form a functional computer. Hence, this report justifies the decision to build an 8-bit CPU from TTL logics as a teaching and learning platform for CSA class in UTP.

Chapter three and four of this dissertation describes the development of the CPU from ground up. The results in these chapters show that a fully functional computer is successfully completed and full comprehension of it - from gate level - was attained and documented.

Future Work

Future work need to be done on the computer would be completing the hardware side by completing the IDE controller. Then the rudimentary operating system designed by Bill Buzbee can be loaded and programs compiled by the C compiler can be run on the machine. Until now, programs in C were compiled but limited to be burnt onto the boot PROM only. Also, as we already have a platform to work on, assembler and compiler can be written for the machine. For the time being, assembler (qas) and the C compiler (retargeted from LCC) are supplied from the designer, Bill Buzbee.

Next leap would be modifying the CPU at the hardware side, writing new microcode, increasing the computer's performance, adding I/O devices (an input device like the keyboard or even a VGA driver maybe), expand the CPU memory capability. There are a lot more things to be done to this CPU comparing it to the computers today (2006).

REFERENCES

Bill Buzbee, 2003 , www.homebrewcpu.com

Morris Mano, Charles Kime, 2004, Logic and computer design fundamentals, Third edition, Pearson Prentice Hall

[1] N. Chang and I. Lee, "Embedded system hardware design course track for CS students," in Proc. IEEE Int. Conf. Microelectronic Systems Education, Anaheim, CA, Jun. 2003, pp. 49-50

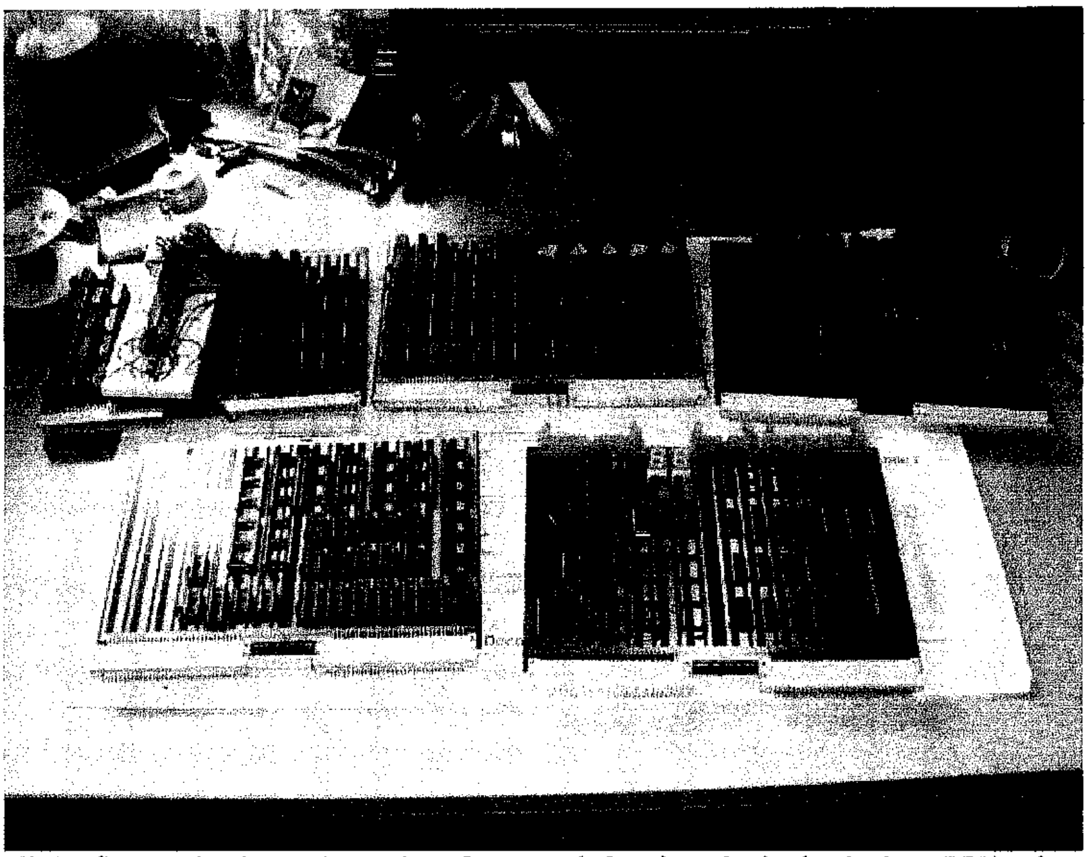
[2] An integrated environment for teaching computer architecture

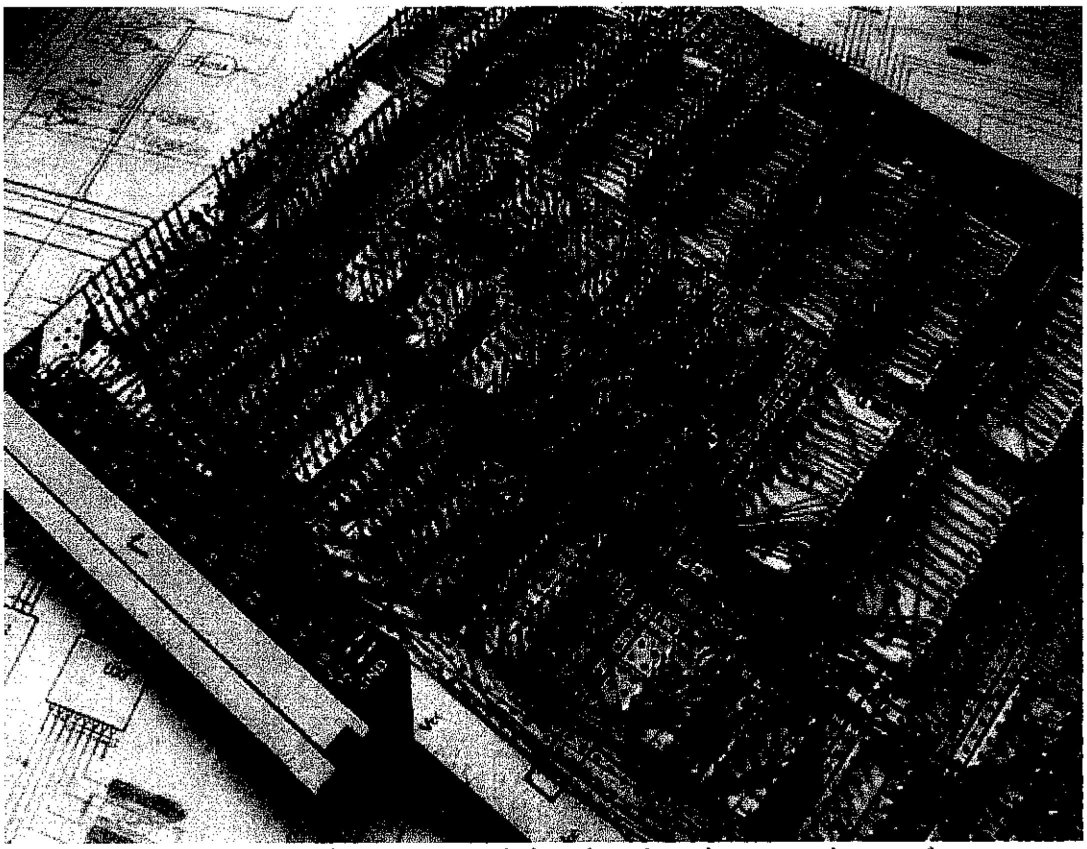
[3] computer.howstuffworks.com

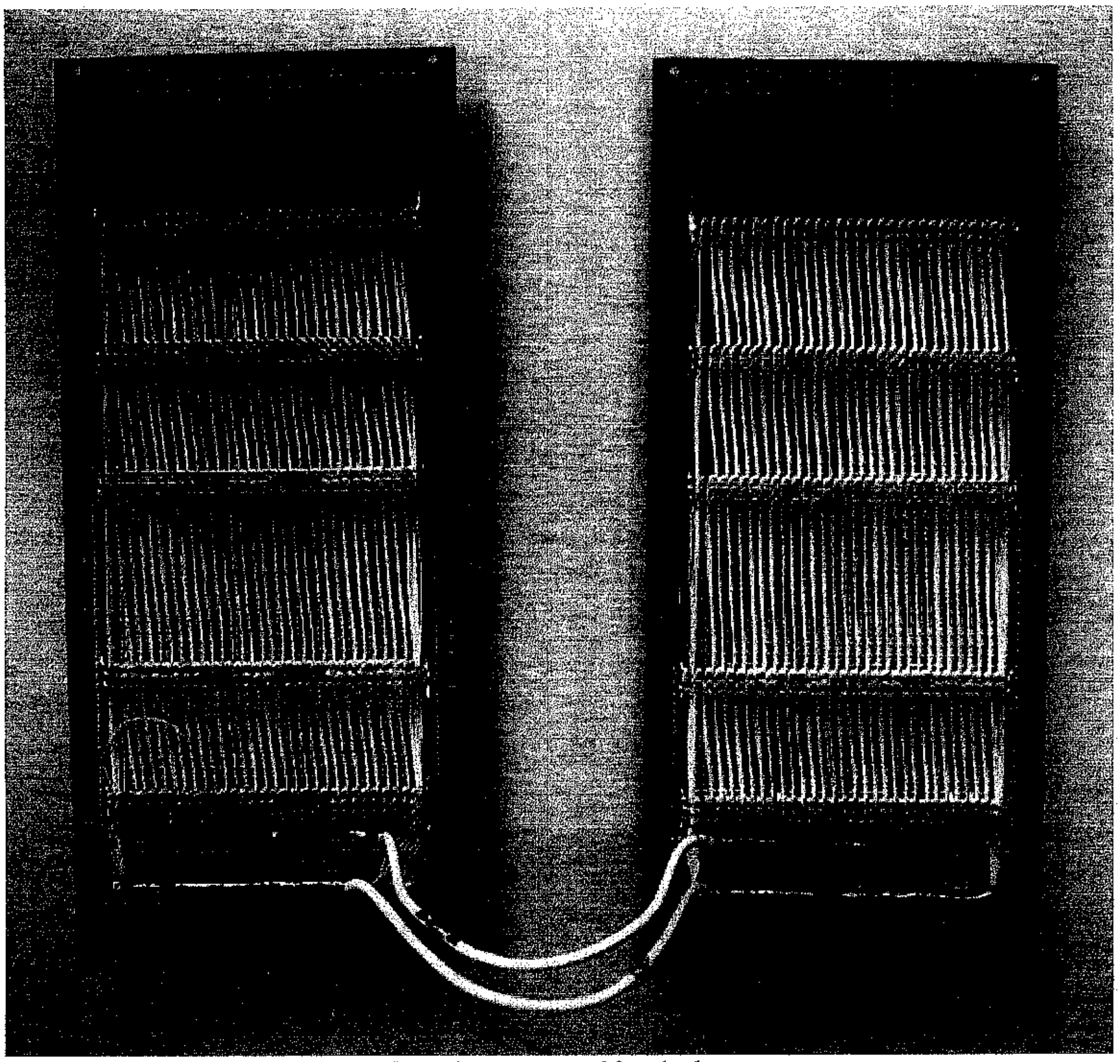
[4] www.wikipedia.com

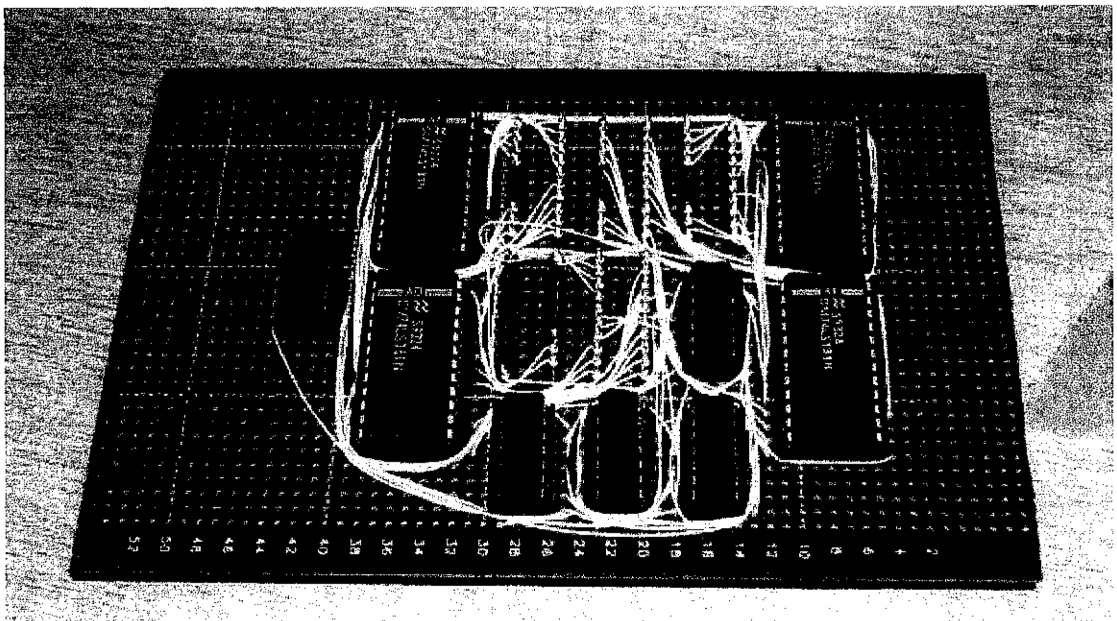
[5] http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibtable.html

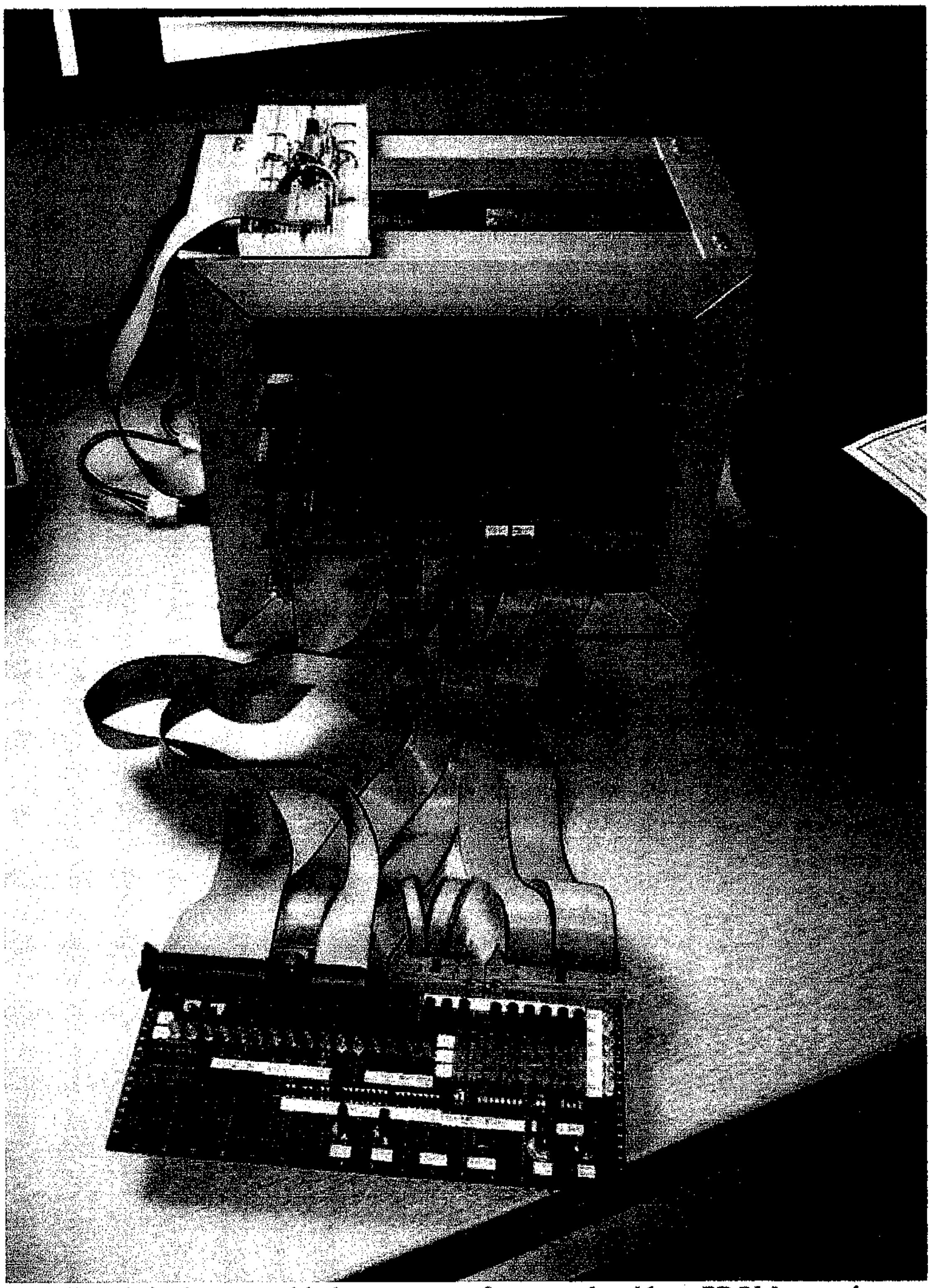
[6] http://www.amigaforever.com/kb/3-105.html

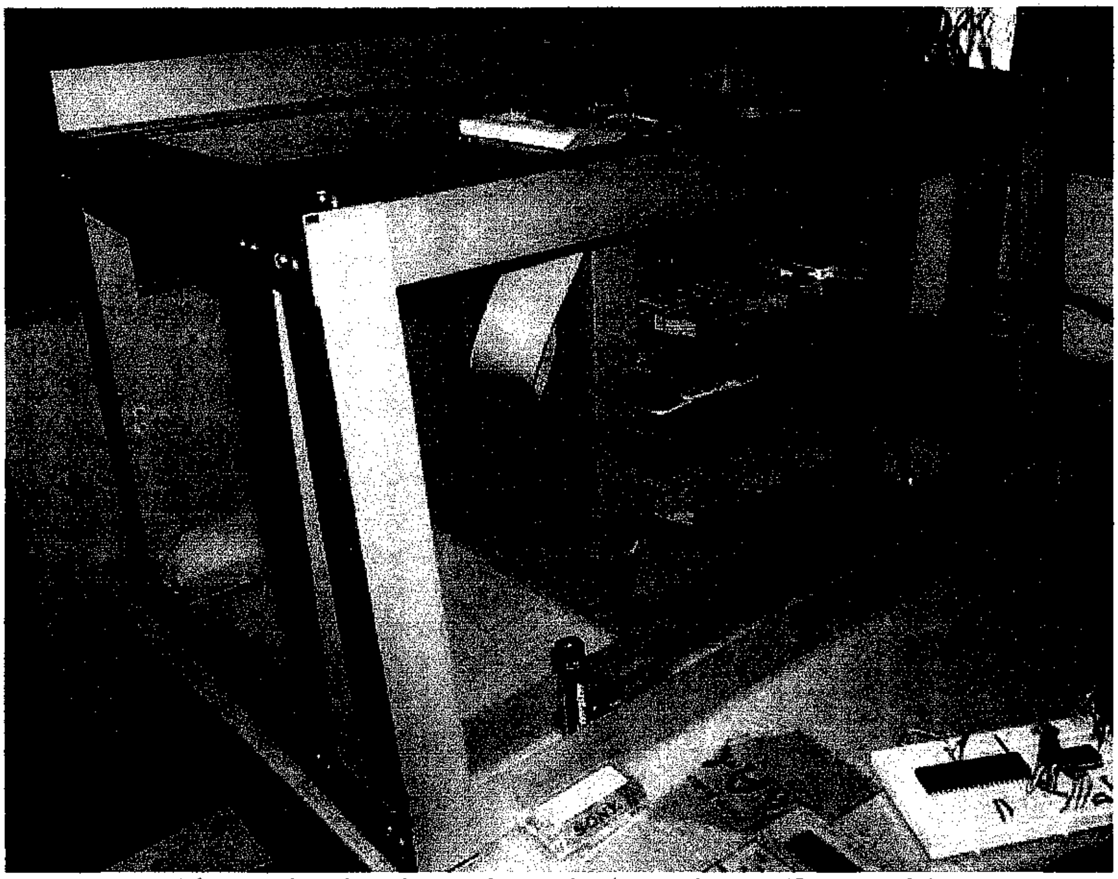

All the five cards pictured together, from top left going clockwise is the ALU/register card, control card, front panel card, device card and memory card. The ALU card is pictured with the breadboard ALU substitute on top.

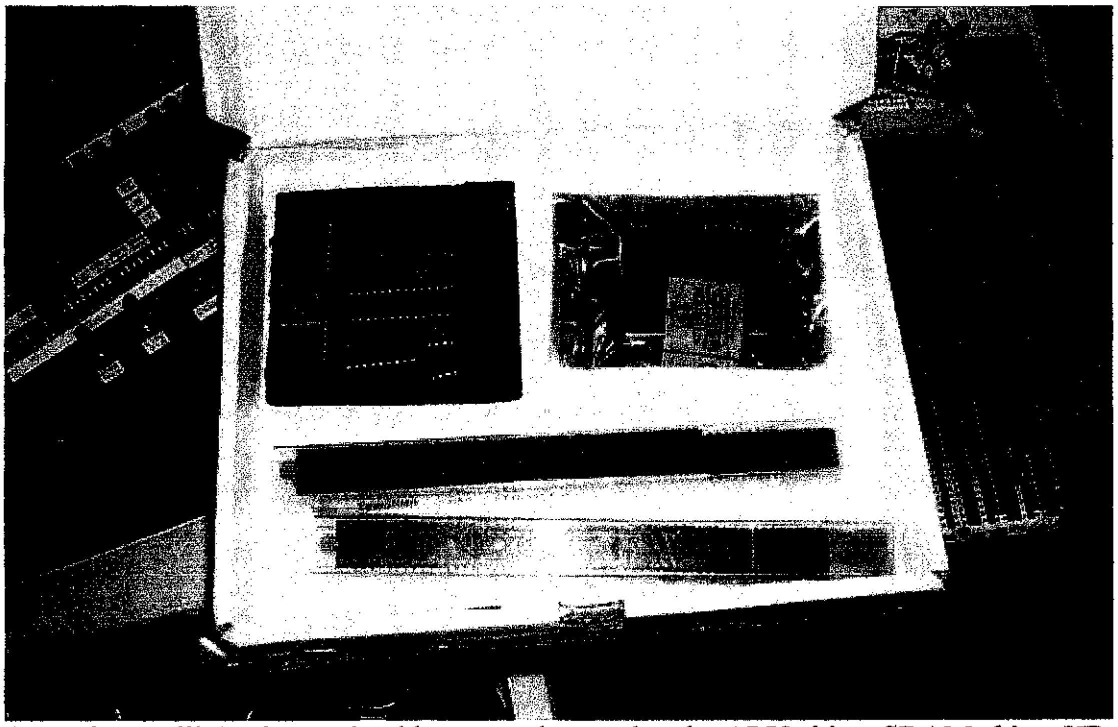

Bottom side of memory card showing the wire-wrapping work

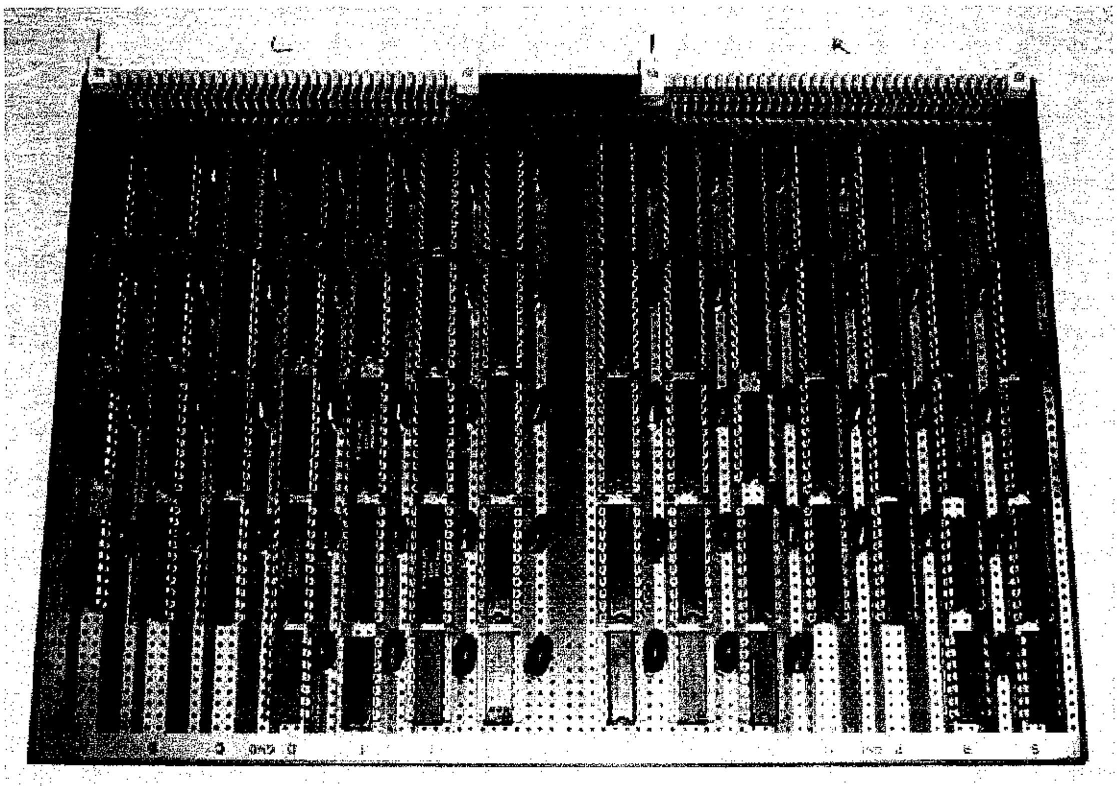

The wire-wrapped backplane

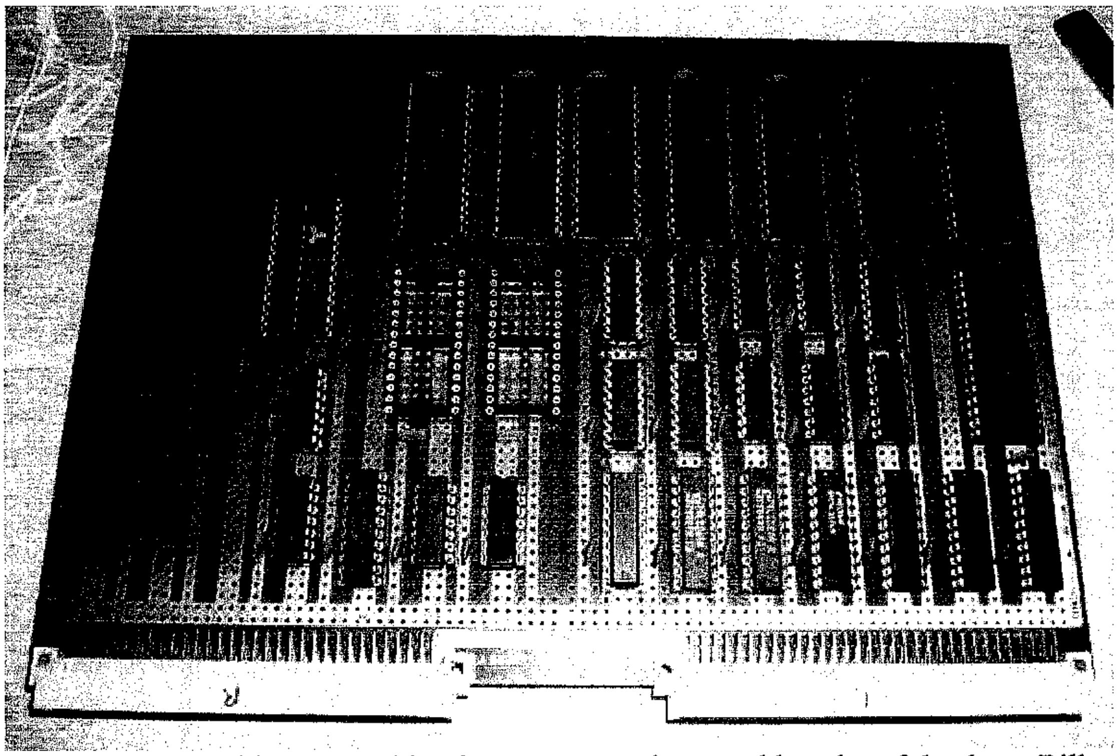

New low-profile ALU substitute board enables all cards to be put into the cage

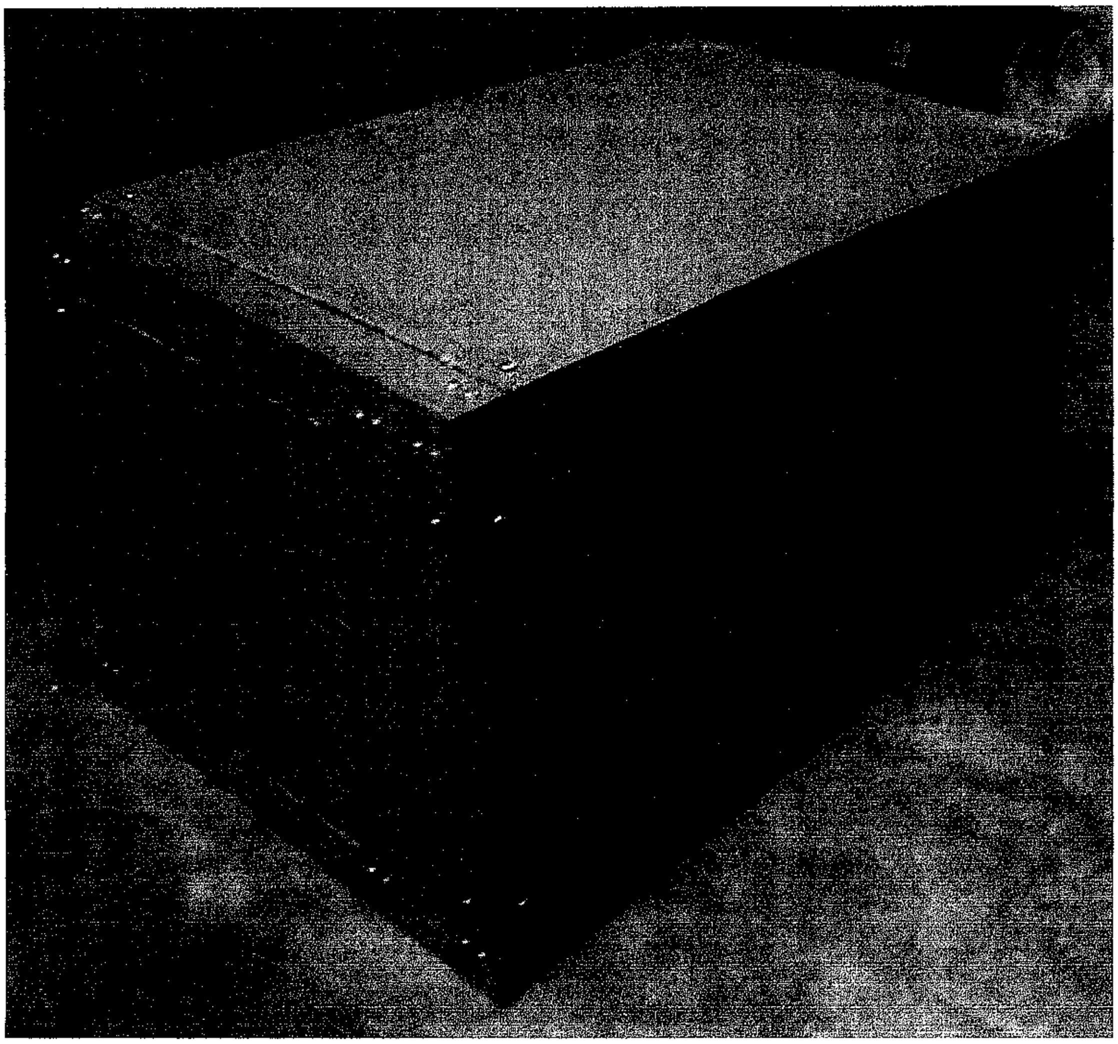

All cards in the cage with the temporary front panel and boot PROM extension connected

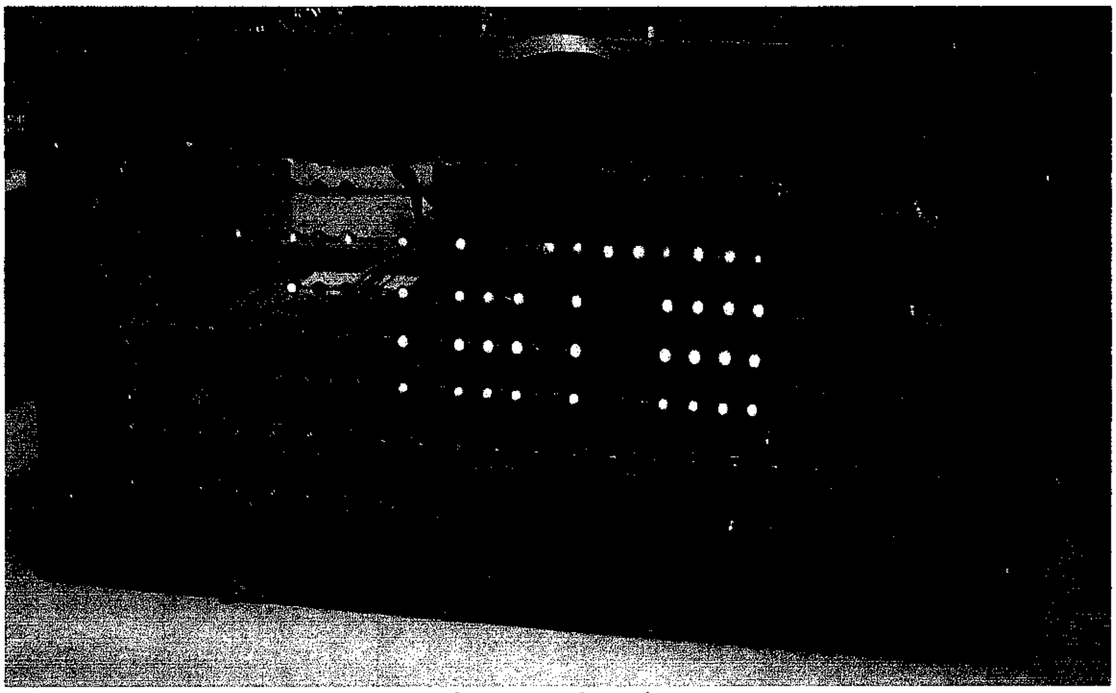

Picture showing the card cage in the enclosure (frame only)

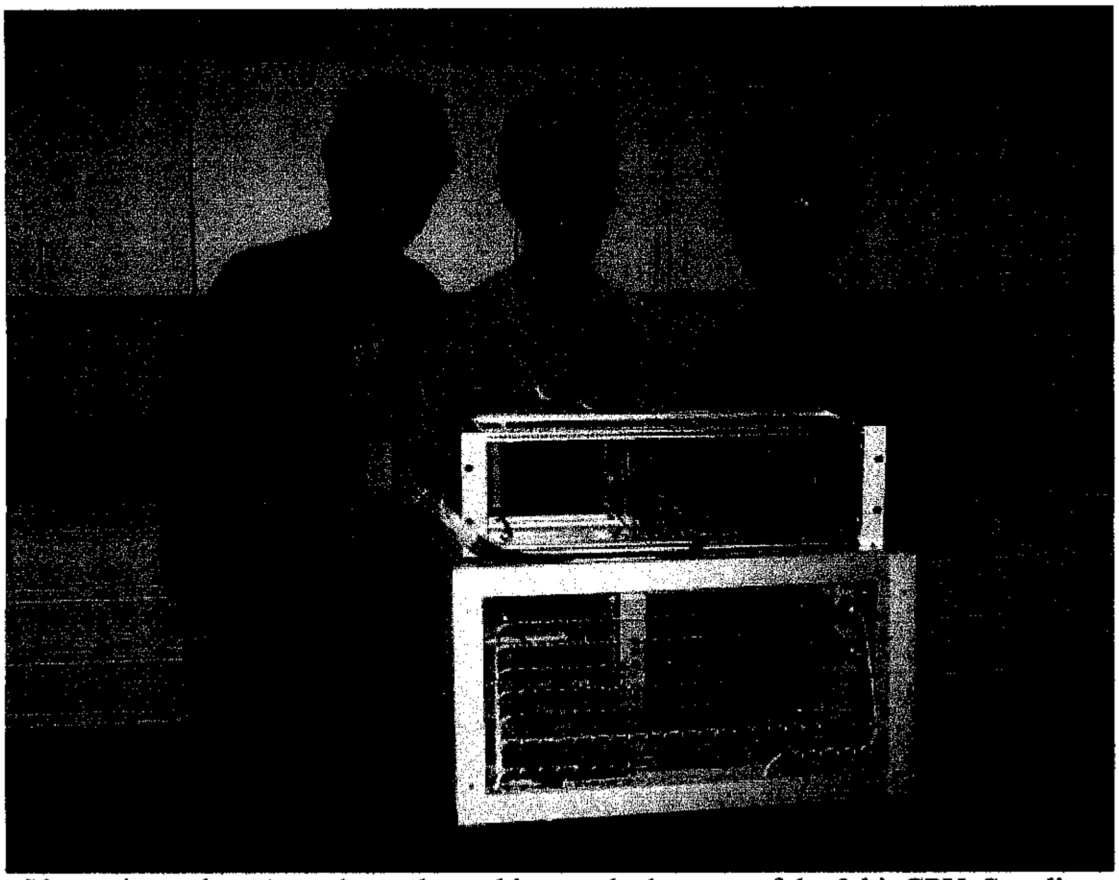

Parts from Bill Buzbee arrived in a parcel, contains the ALU chips, SRAM chips, HP HEX display, real time clock (RTC) chip, UART chips, digital delay devices and a source CD.

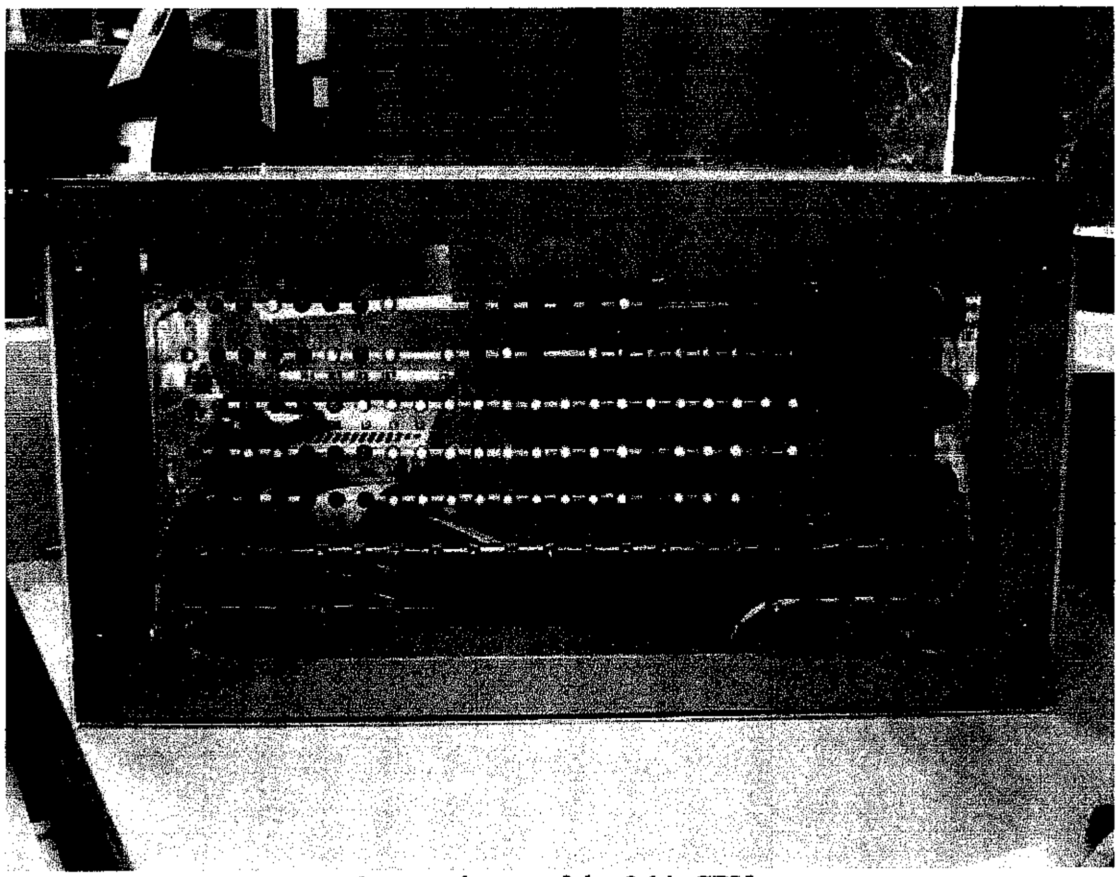

The ALU card now populated with the exact chips, no more substitute board, ALU chips courtesy of Bill Buzbee


Memory card with SRAM chips for memory and page table, a lot of thanks to Bill Buzbee for the parts


The enclosure in progress


New front panel on the case


Pictured together, Aaron's stack machine stacked on top of the 8-bit CPU. Standing from left, Aaron who worked on the Mark 1 Forth computer, Dr Yap our supervisor and me.


Latest picture of the 8-bit CPU.

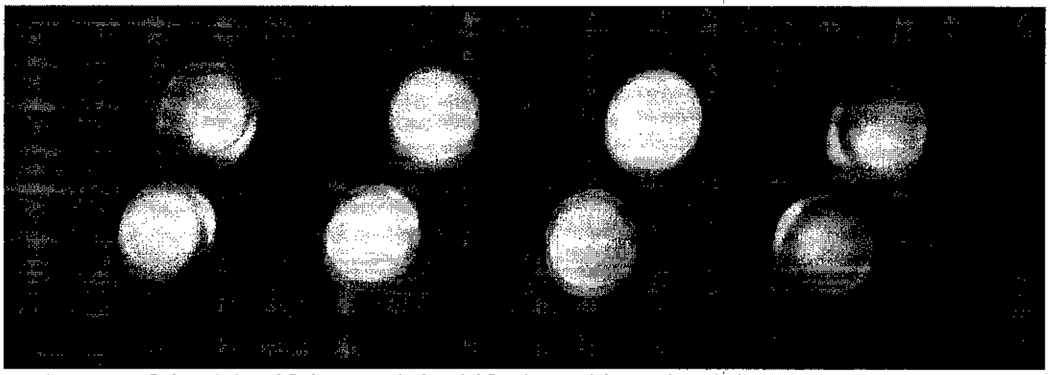
Select			Operation
S_0	S_1	S_2	
L	L	L	Clear
H	L	L	B Minus A
L	H	L	A Minus B
H	H	L	A Plus B
L	L	H	A⊕B
H	L	H	A + B
L	H	H	AB
H	H	H	Preset

H = HIGH Voltage Level
L = LOW Voltage Level

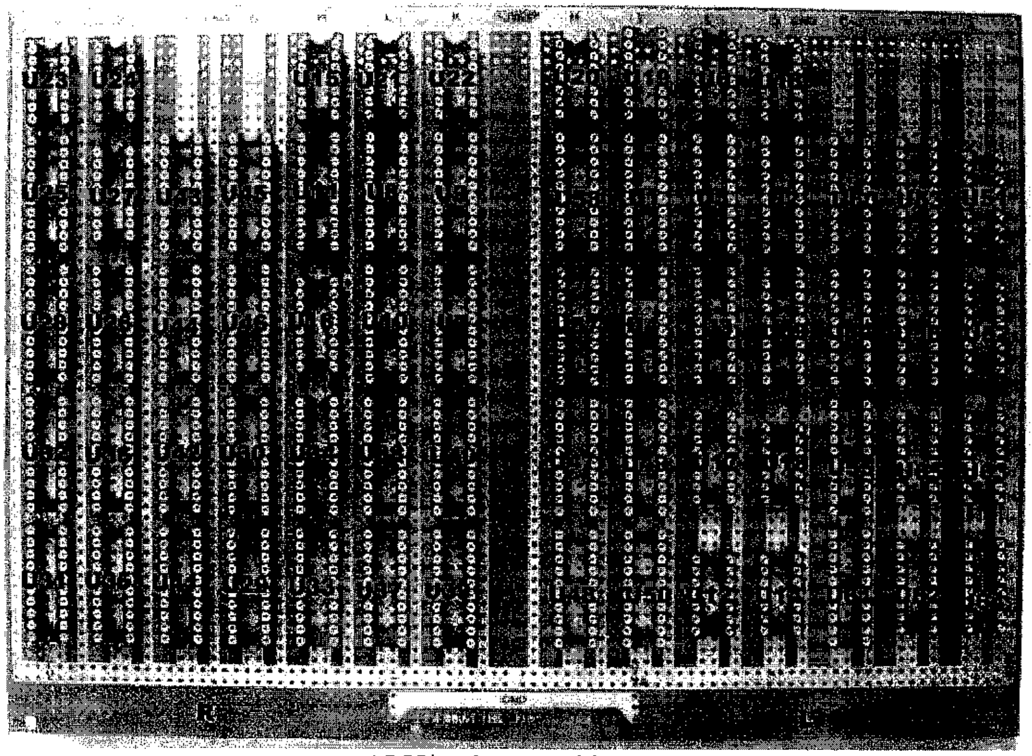
74381 and 74381 function select table

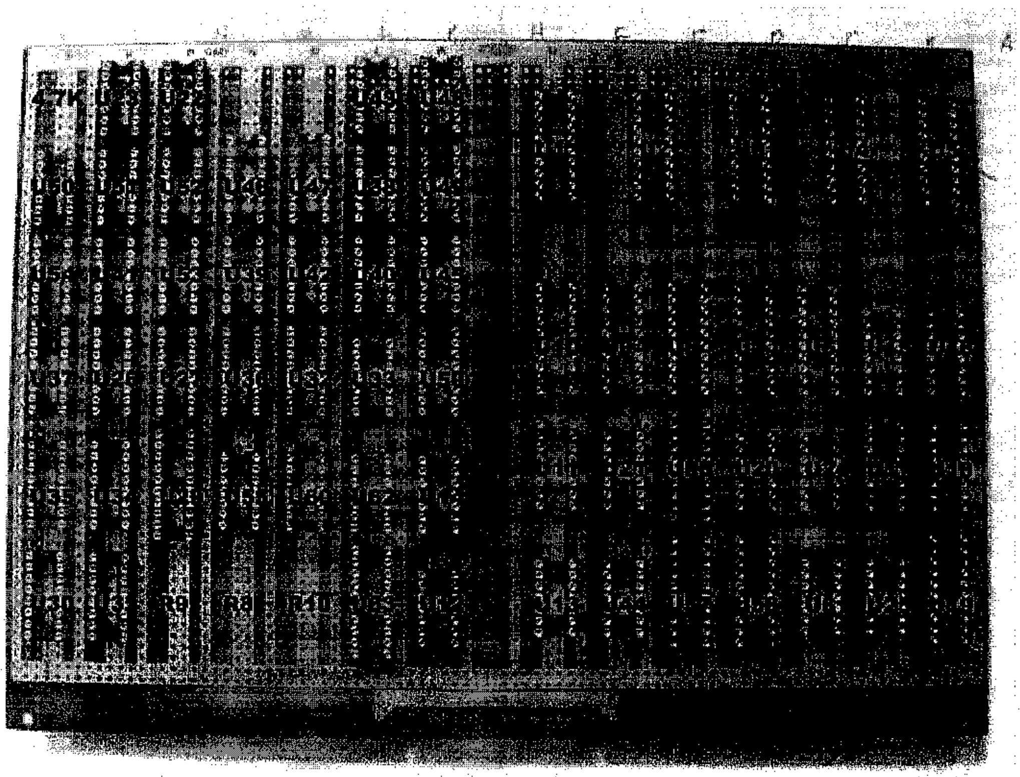
Mode Select Inputs				Active LOW Operands & F_n Outputs		Active HIGH Operands & F_n Outputs	
S3	S2	S1	S0	Logic (M = H)	Arithmetic** (M = L) (C_n = L)	Logic (M = H)	Arithmetic** (M = L) (C_n = H)
L	L	L	L	$\overline{A}$	A minus 1	$\overline{A}$	A
L	L	L	H	$\overline{AB}$	AB minus 1	$\overline{A + B}$	A + B
L	L	H	L	$\overline{A + B}$	$A\overline{B}$ minus 1	$\overline{A}B$	$A + \overline{B}$
L	L	H	H	Logic 1	minus 1	Logic 0	minus 1
L	H	L	L	$\overline{A + B}$	A plus $(A + \overline{B})$	$\overline{AB}$	A plus $A\overline{B}$
L	H	L	H	$\overline{B}$	AB plus $(A + \overline{B})$	$\overline{B}$	(A + B) plus $A\overline{B}$
L	H	H	L	$\overline{A \oplus B}$	A minus B minus 1	$A \oplus B$	A minus B minus 1
L	H	H	H	$A + \overline{B}$	$A + \overline{B}$	$A\overline{B}$	AB minus 1
H	L	L	L	$\overline{A}B$	A plus (A + B)	$\overline{A} + B$	A plus AB
H	L	L	H	$A \oplus B$	A plus B	$\overline{A \oplus B}$	A plus B
H	L	H	L	B	$A\overline{B}$ plus (A + B)	B	$(A + \overline{B})$ plus AB
H	L	H	H	A + B	A + B	AB	AB minus 1
H	H	L	L	Logic 0	A plus A*	Logic 1	A plus A*
H	H	L	H	$A\overline{B}$	AB plus A	$A + \overline{B}$	(A + B) plus A
H	H	H	L	AB	$A\overline{B}$ minus A	A + B	$(A + \overline{B})$ plus A
H	H	H	H	A	A	A	A minus 1

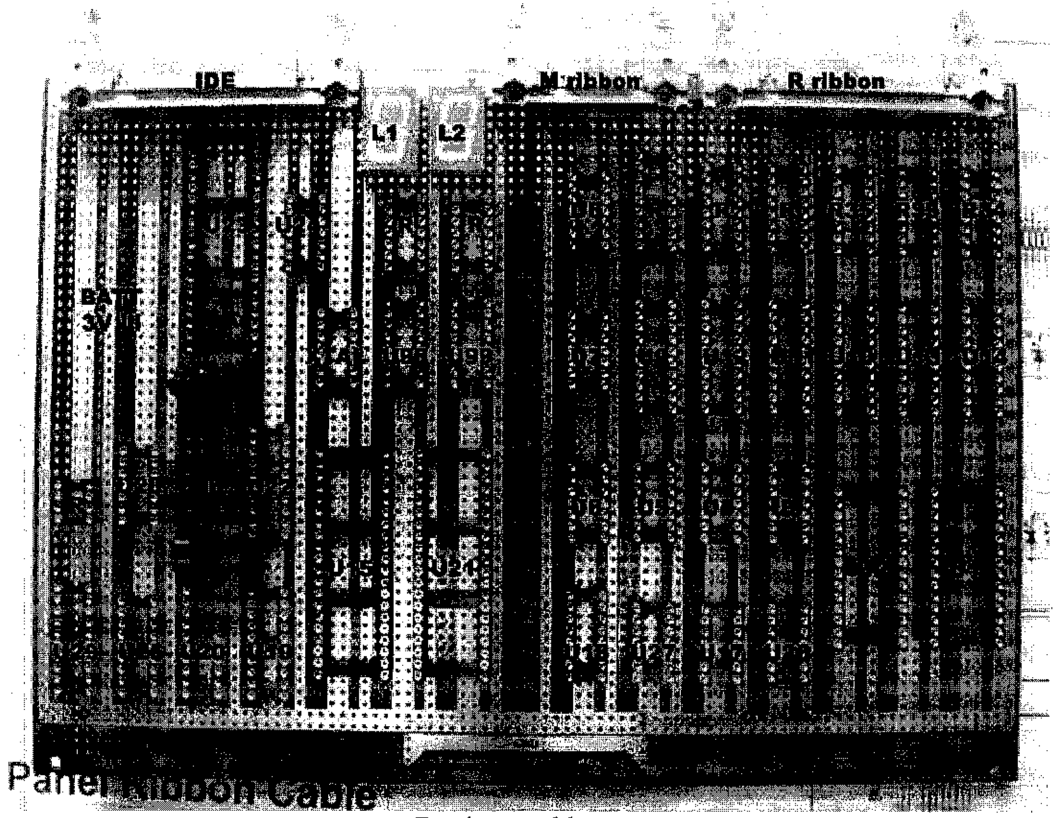
74181 function select table


Output of the ALU if the result is shifted one bit to the right (_DO_RSHIFT = 0)

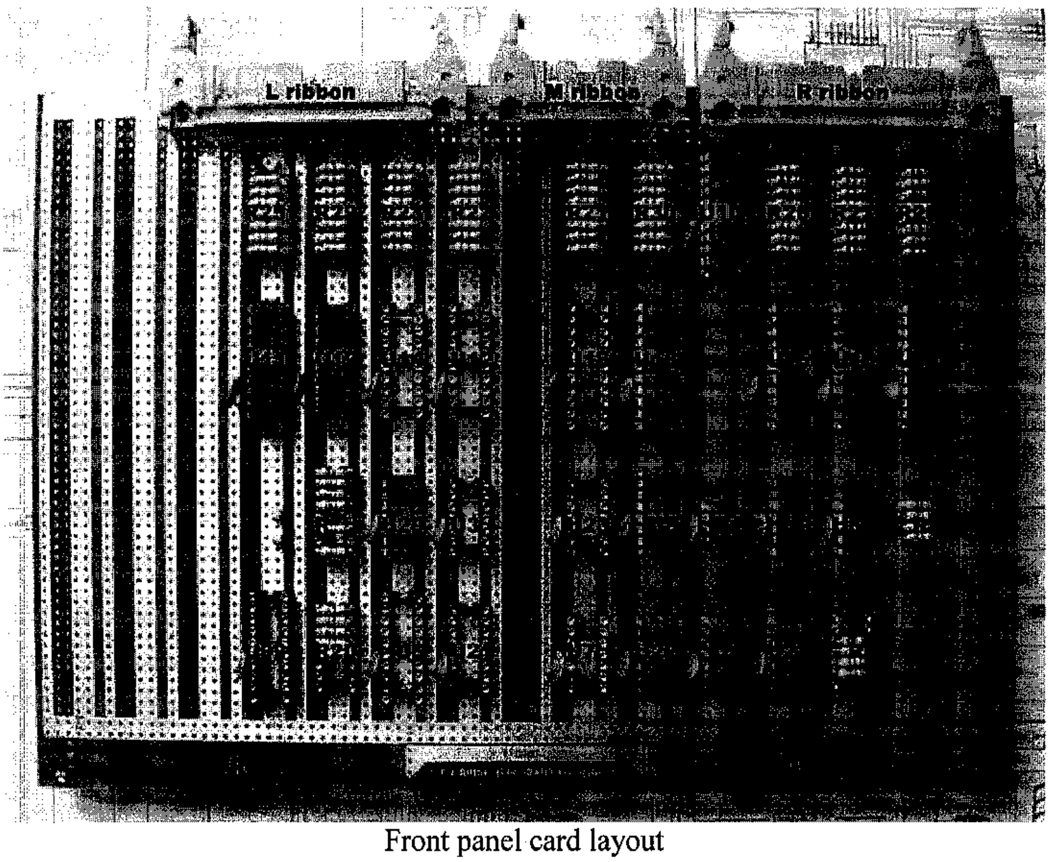
Appendix I – board layouts and backplane pinout


ALU/register card layout

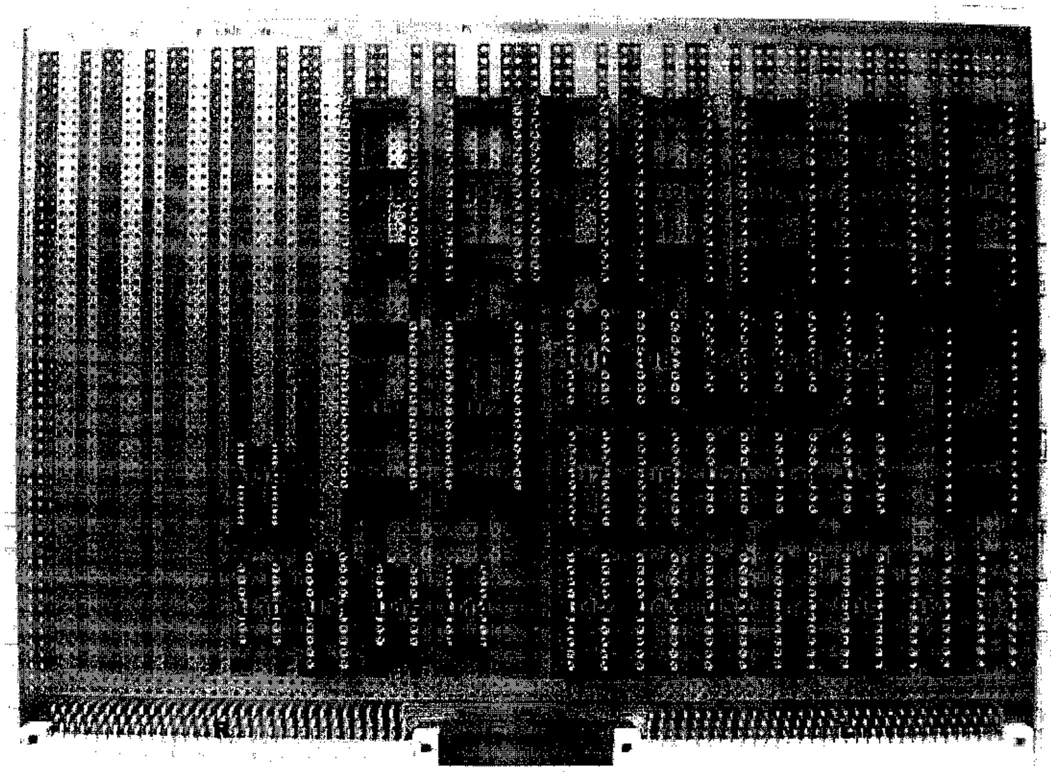

Control card layout

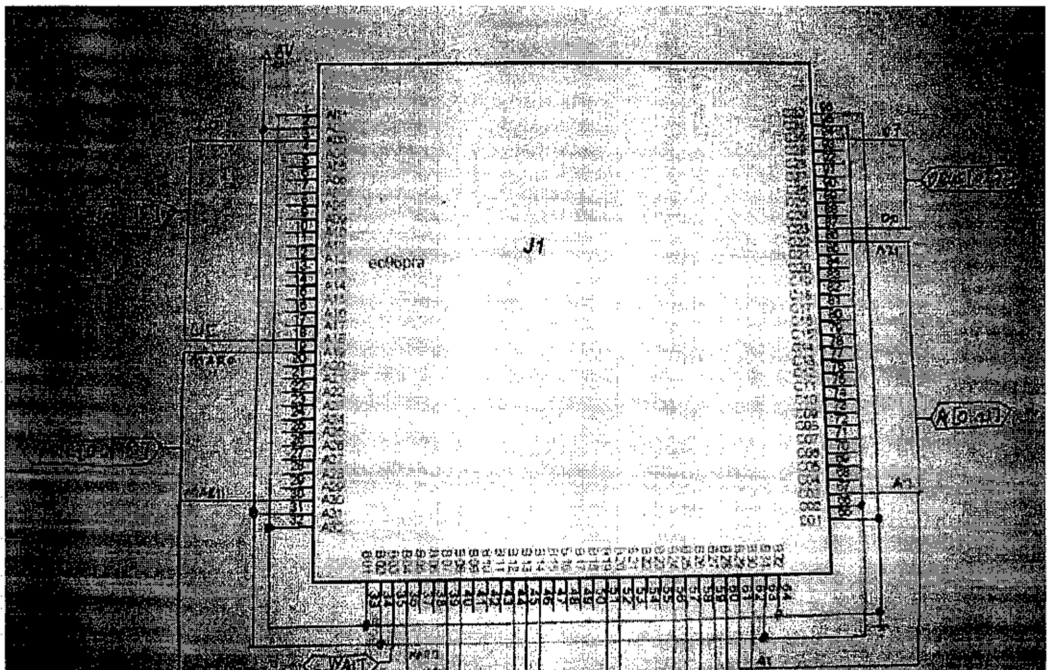


Device card layout

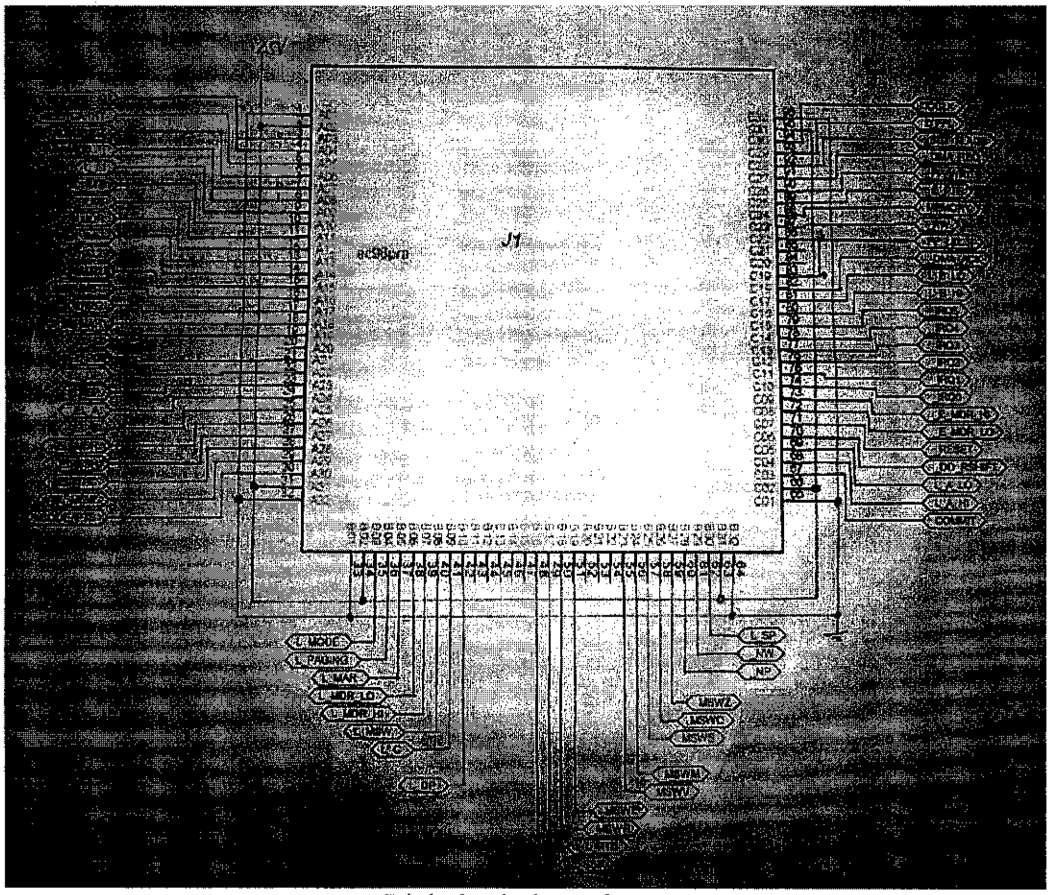


Front panel card layout


Memory card layout

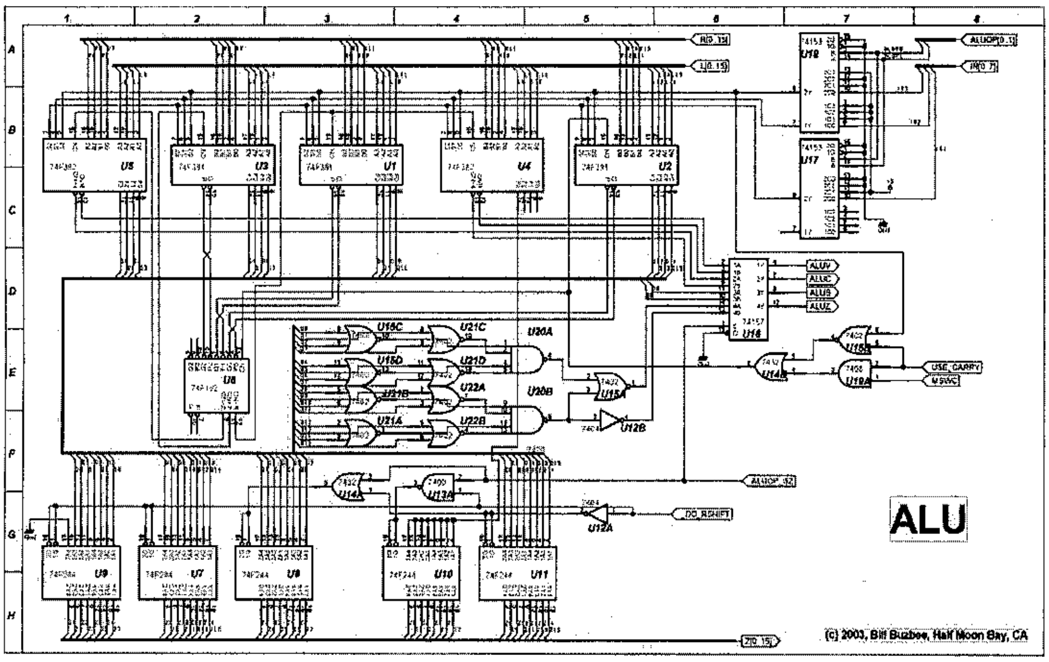


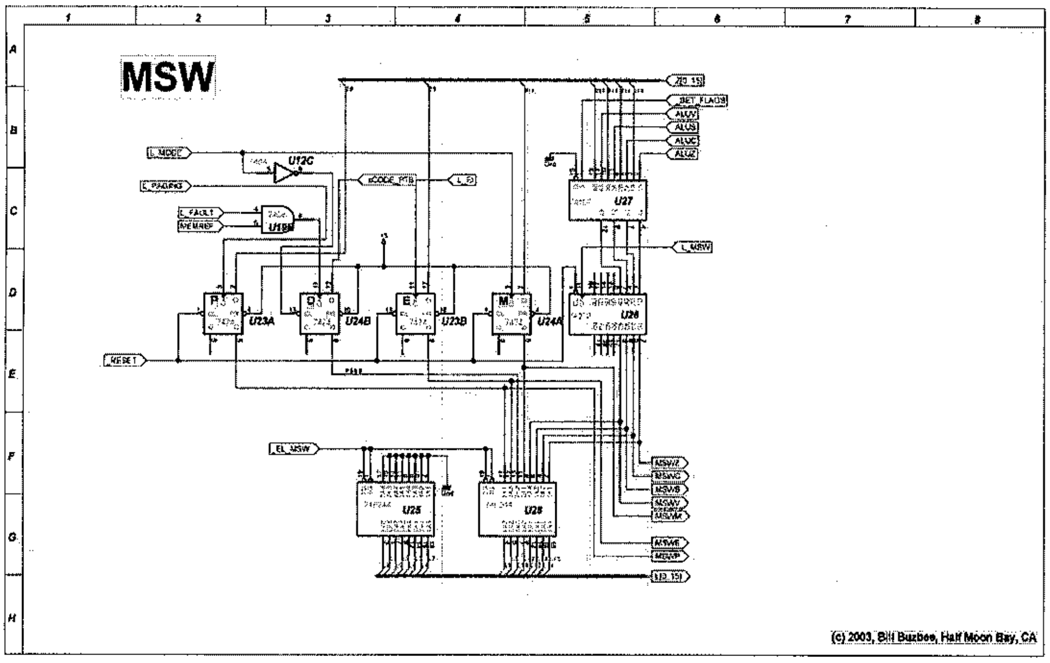
Left backplane pinout

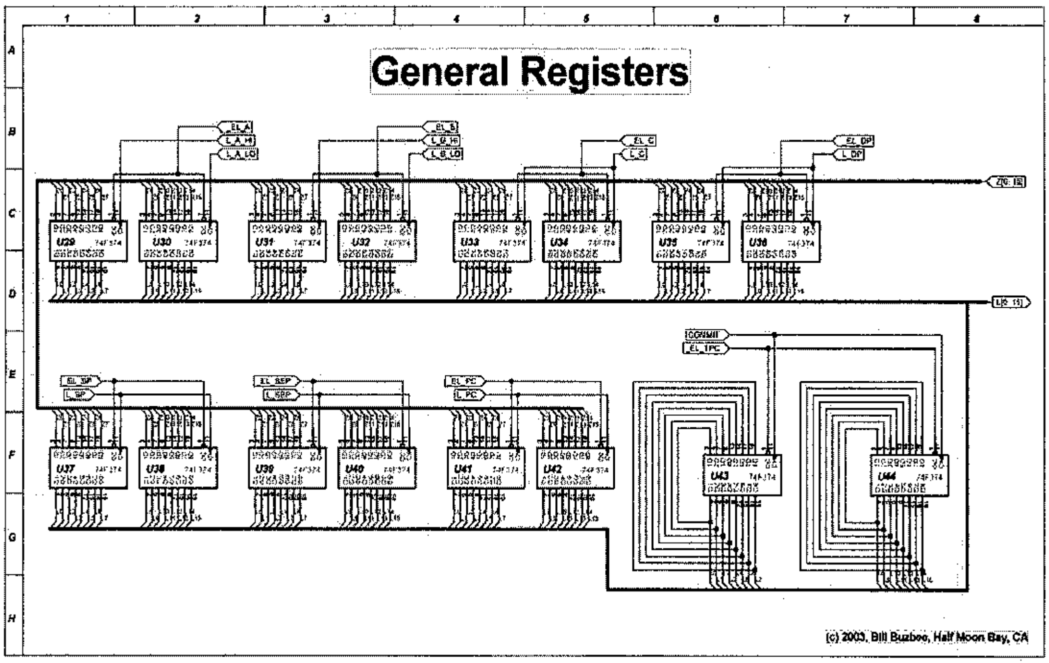


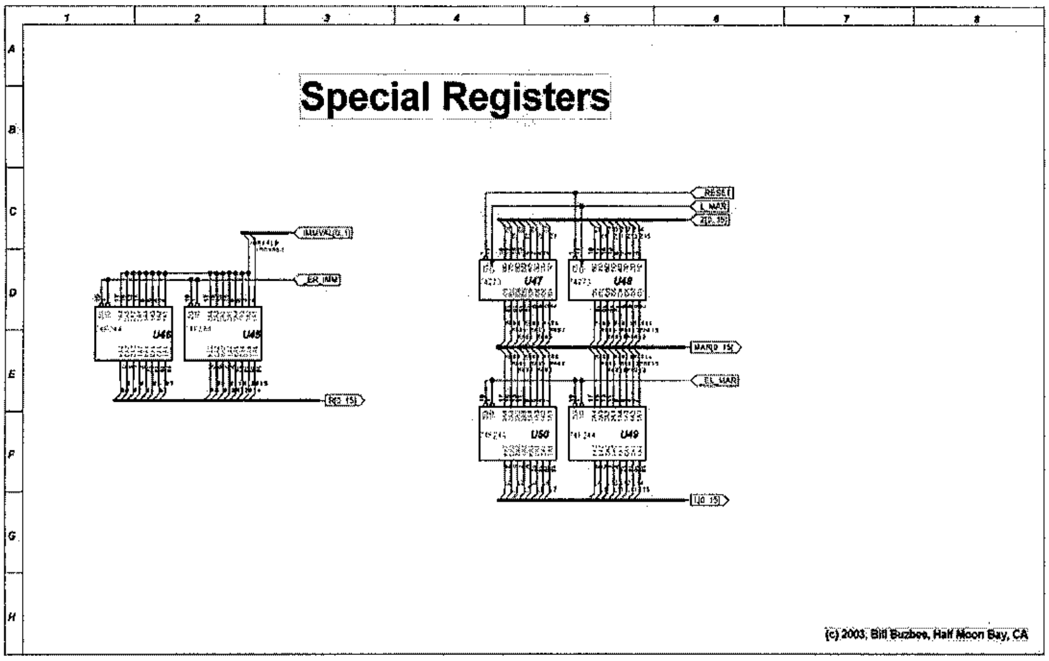
Right backplane pinout

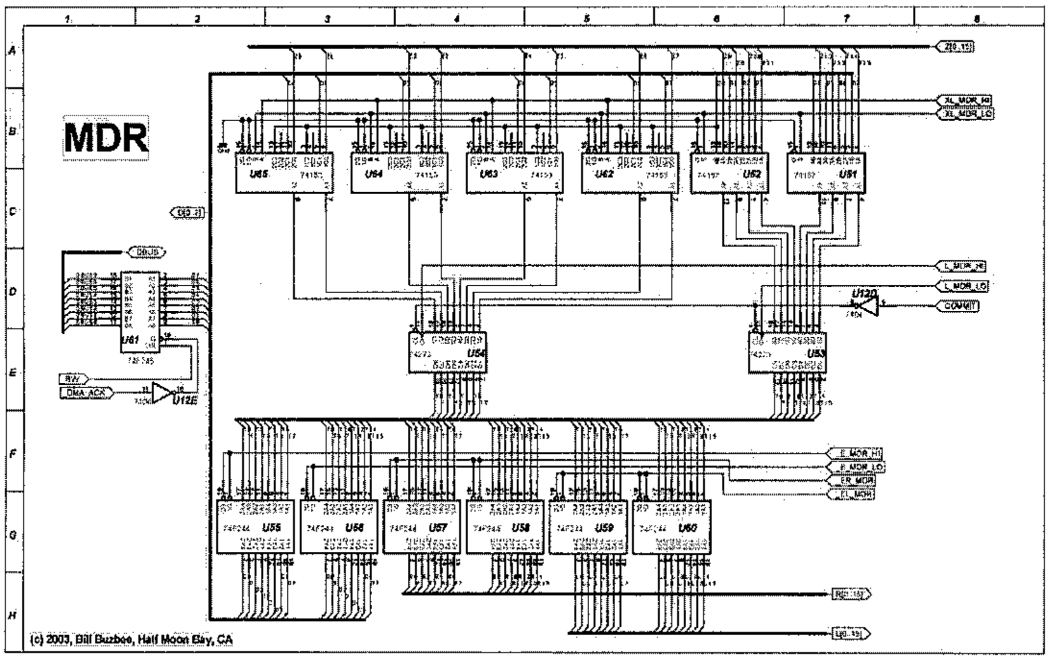
Appendix II – schematic diagrams from www.homebrewcpu.com

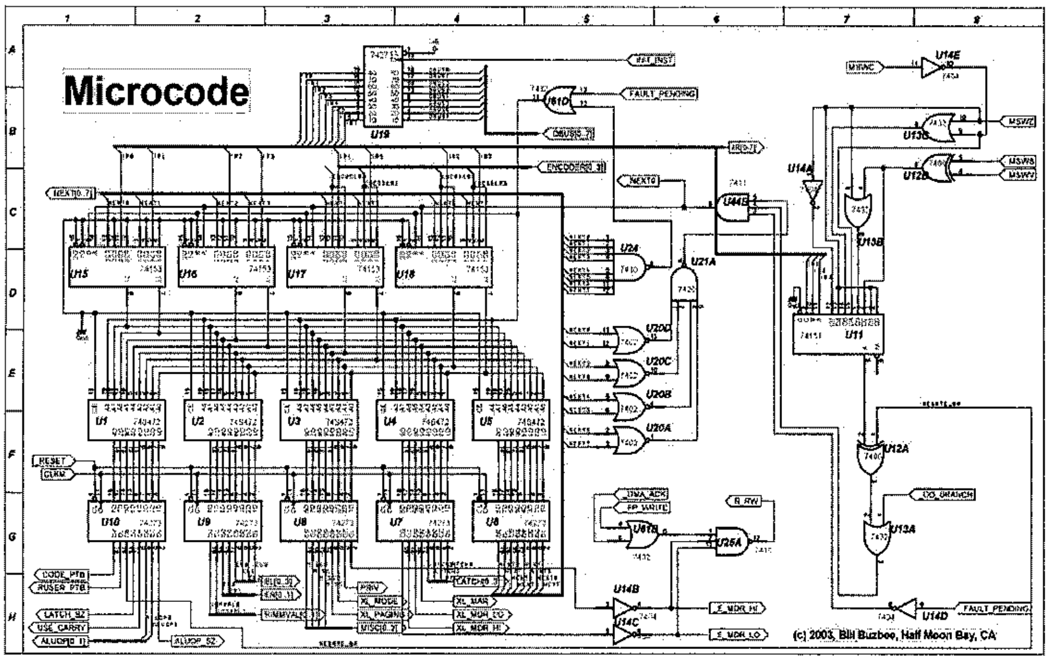


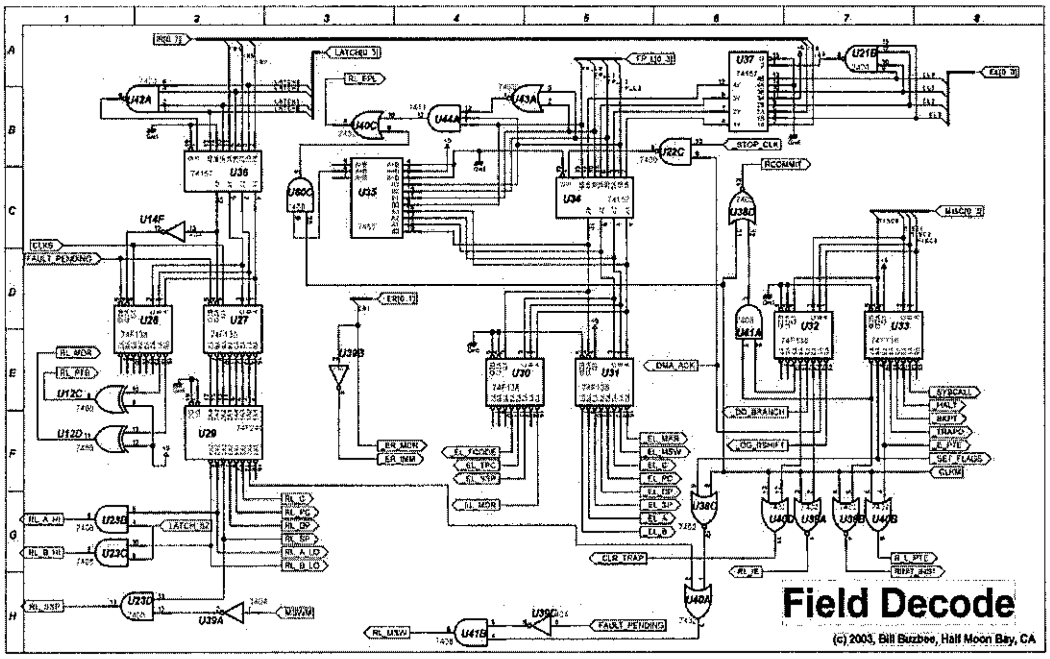


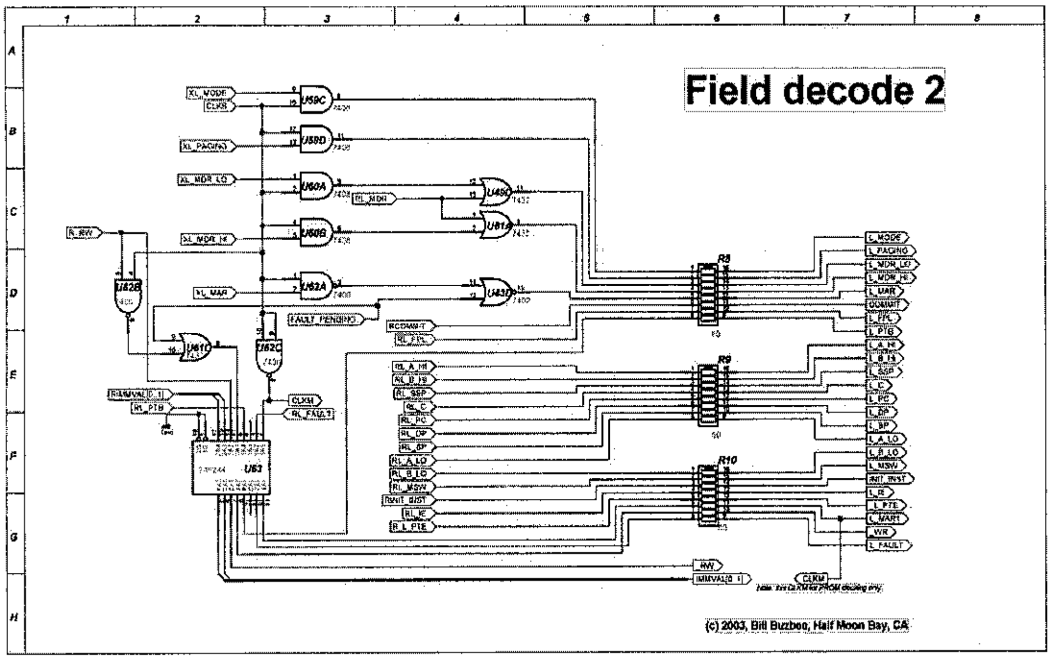

General Registers
EL_A
L_A_HI
L_A_LO
EL_B
L_B_HI
L_B_LO
EL_C
L_C
EL_DP
L_DP
Z[0..15]
U29
U30
U31
U32
U33
U34
U35
U36
L[0..15]
COMMIT
EL_TPC
EL_SP
L_SP
EL_SSP
L_SSP
EL_PC
L_PC
U37
U38
U39
U40
U41
U42
U43
U44
74F374
(c) 2003, Bill Buzbee, Half Moon Bay, CA

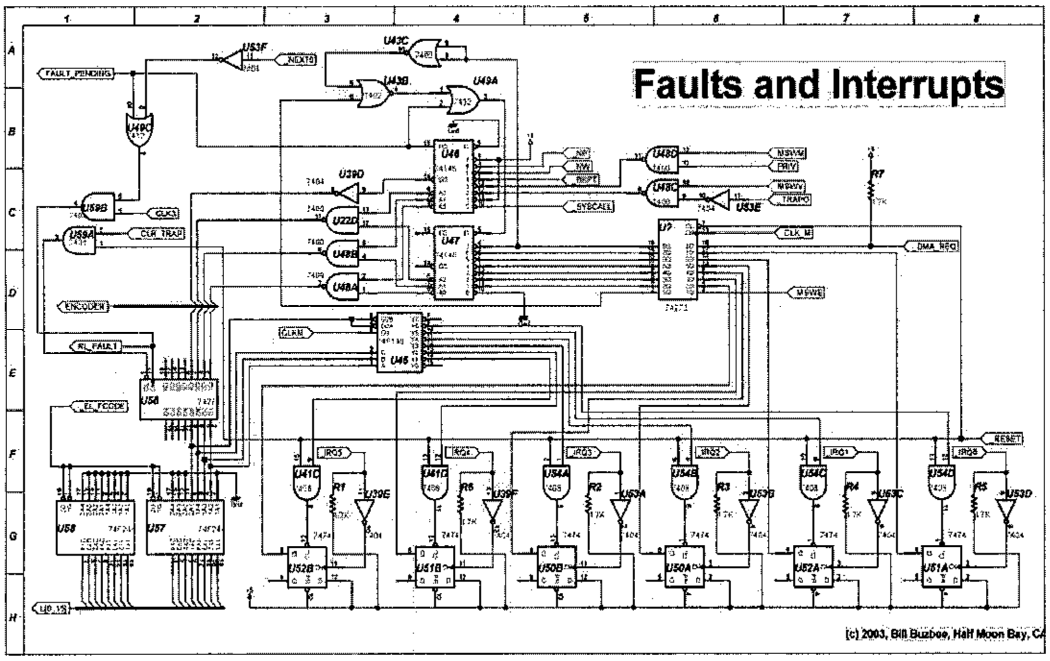

Special Registers
RESET
L_MAR
Z[0..15]
MUVAL[0..1]
ER_IMM
U46
U45
R[0..15]
U47
U48
MAR[0..15]
EL_MAR
U50
U49
L[0..15]
74F244
(c) 2003, Bill Buzbee, Half Moon Bay, CA


MDR
U65
U64
U63
U62
U52
U51
U61
U54
U53
U12D
U12E
U55
U56
U57
U58
U59
U60
Z[0..15]
XL_MDR_HI
XL_MDR_LO
L_MDR_HI
L_MDR_LO
COMMIT
DBUS
RW
DMA_ACK
E_MDR_HI
E_MDR_LO
ER_MDR
EL_MDR
R[0..15]
L[0..15]
(c) 2003, Bill Buzbee, Half Moon Bay, CA

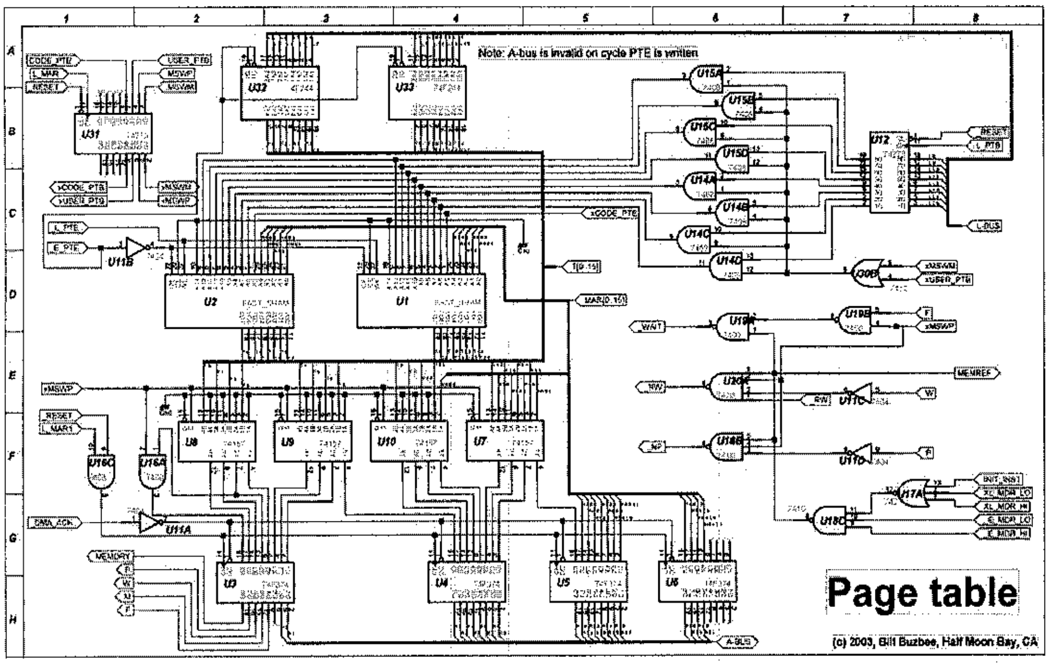

Microcode
U19
U61D
U15
U16
U17
U18
U1
U2
U3
U4
U5
U10
U9
U8
U7
U6
U24
U21A
U20D
U20C
U20B
U20A
U44B
U14A
U14E
U13B
U12B
U11
U12A
U13A
U61B
U25A
U14B
U14C
U14D
INIT_INST
FAULT_PENDING
DBUS[0..7]
IR[0..7]
ENCODER[0..3]
NEXT0
NEXT[0..7]
MSWC
MSWZ
MSWS
MSWV
RESET
CLKM
DMA_ACK
FP_WRITE
R_RW
DO_BRANCH
CODE_PTB
RUSER_PTB
LATCH_SZ
USE_CARRY
ALUOP[0..1]
ALUOP_SZ
PRIV
XL_MODE
XL_PAGING
MISC[0..3]
XL_MAR
XL_MDR_LO
XL_MDR_HI
E_MDR_HI
E_MDR_LO
IMMVAL[0..3]
(c) 2003, Bill Buzbee, Half Moon Bay, CA


Field Decode
(c) 2003, Bill Buzbee, Half Moon Bay, CA

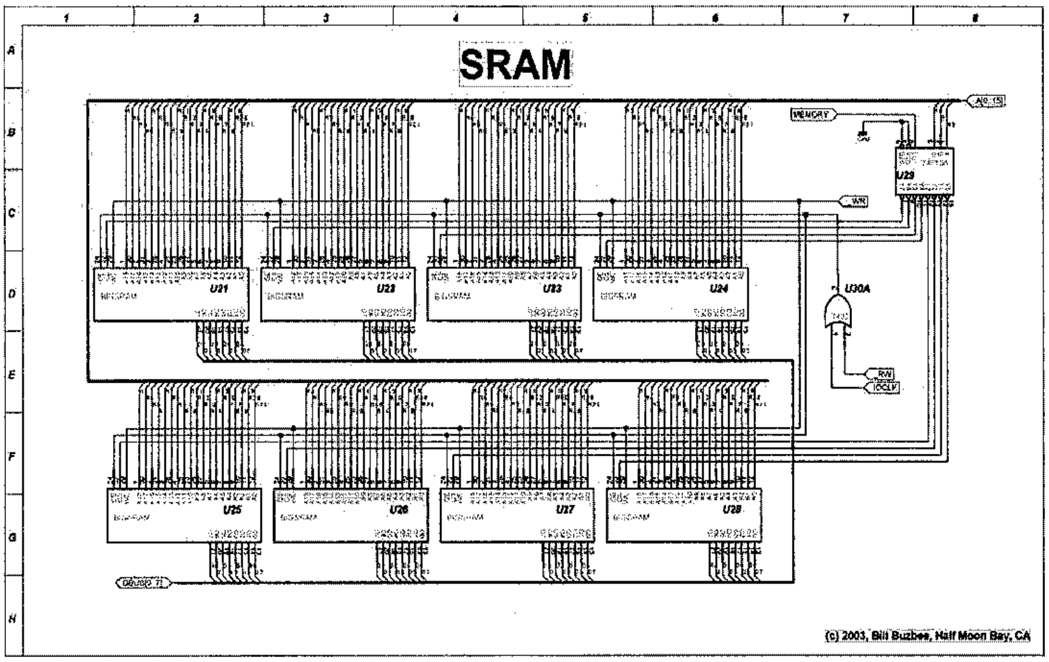

Field decode 2
(c) 2003, Bill Buzbee, Half Moon Bay, CA

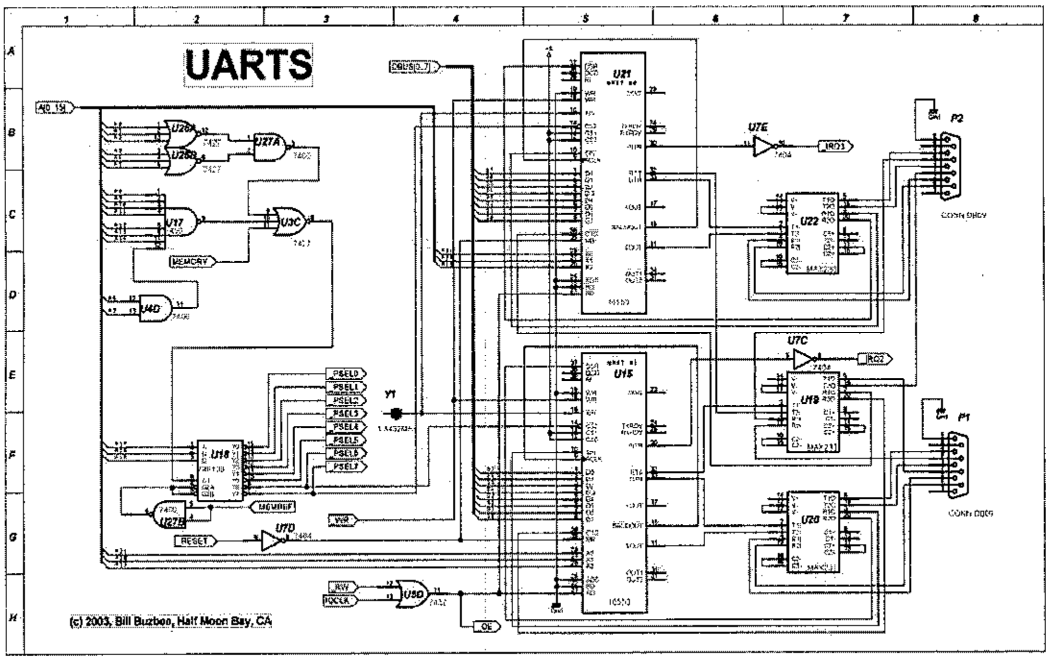


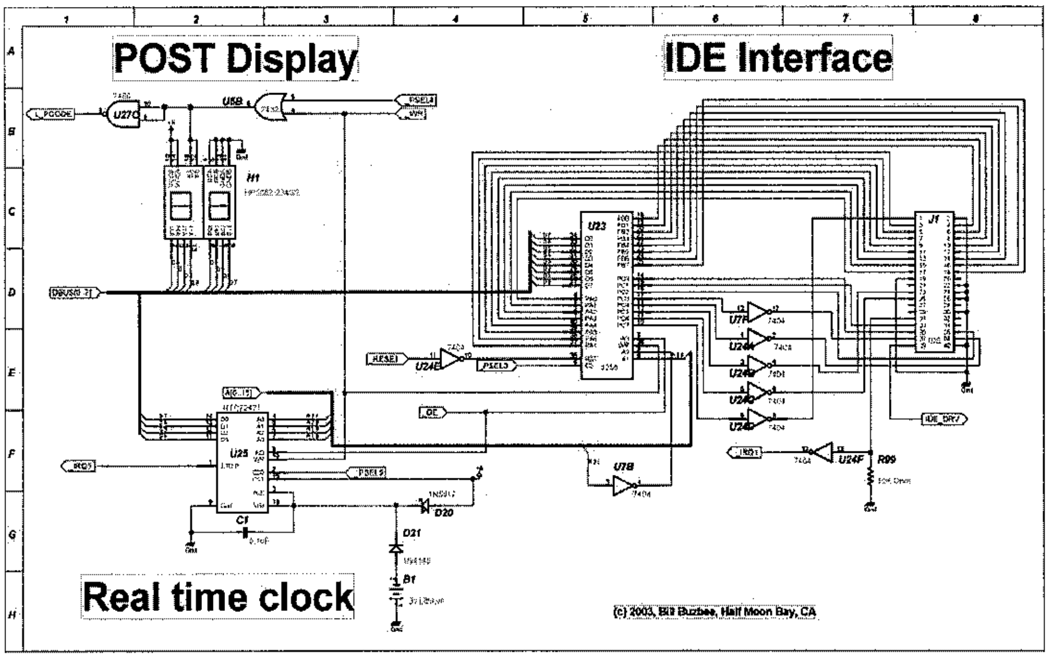
Faults and Interrupts
FAULT_PENDING
NEXT0
ENCODER
EL_FAULT
EL_FCODE
RESET
IRQ5
IRQ4
IRQ3
IRQ2
IRQ1
IRQ0
MSWM
PRIV
MSWV
TRAPO
SYSCALL
CLK_M
DMA_REQ
MSWE
CLKS
CLR_TRAP
CLKM
U0..15
(c) 2003, Bill Buzbee, Half Moon Bay, CA

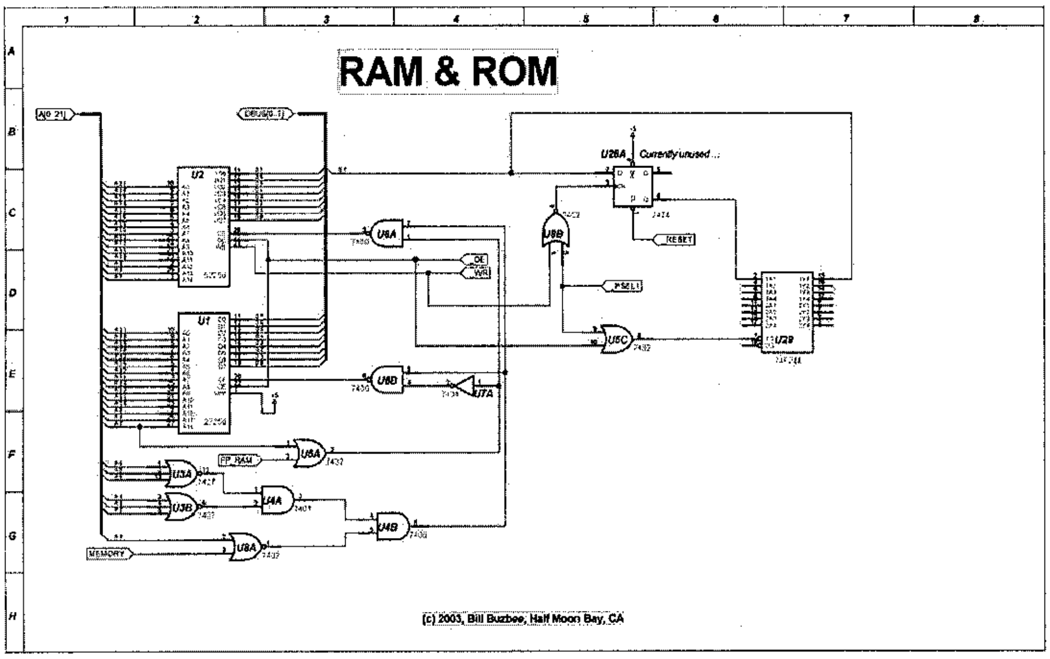


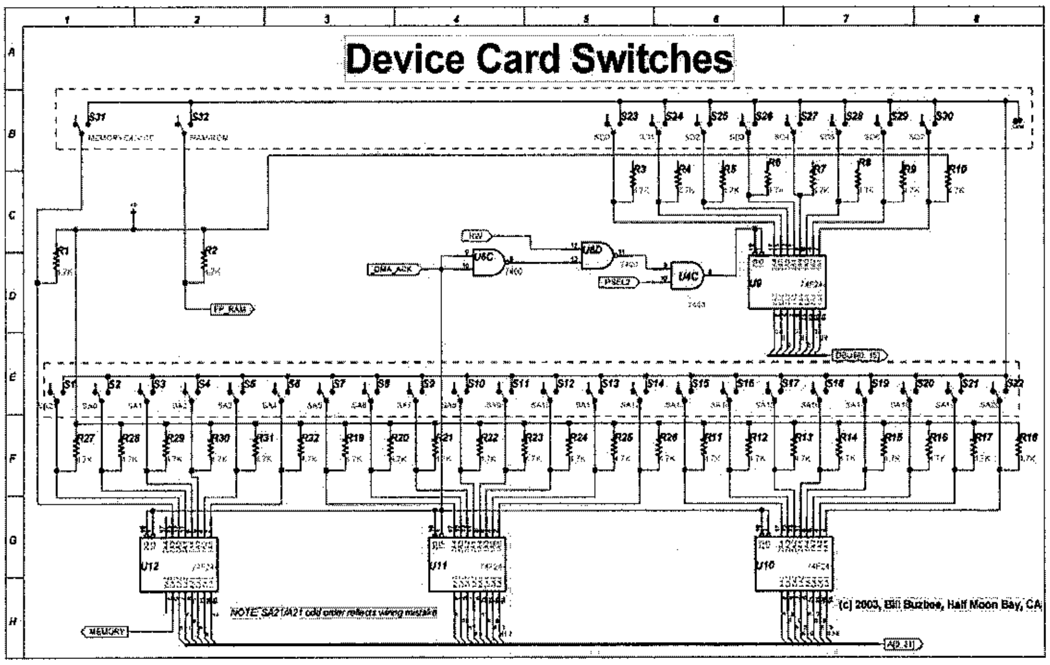
Note: A-bus is invalid on cycle PTE is written
CODE_PTB
L_MAR
RESET
USER_PTB
MSWP
MSWM
xCODE_PTB
xUSER_PTB
xMSWM
xMSWP
L_PTE
E_PTE
xCODE_PTE
T[0..15]
MAR[0..15]
WAIT
RW
NP
MEMREF
xMSWP
DMA_ACK
MEMORY
L_PTB
L-BUS
A-BUS
Page table
(c) 2003, Bill Buzbee, Half Moon Bay, CA

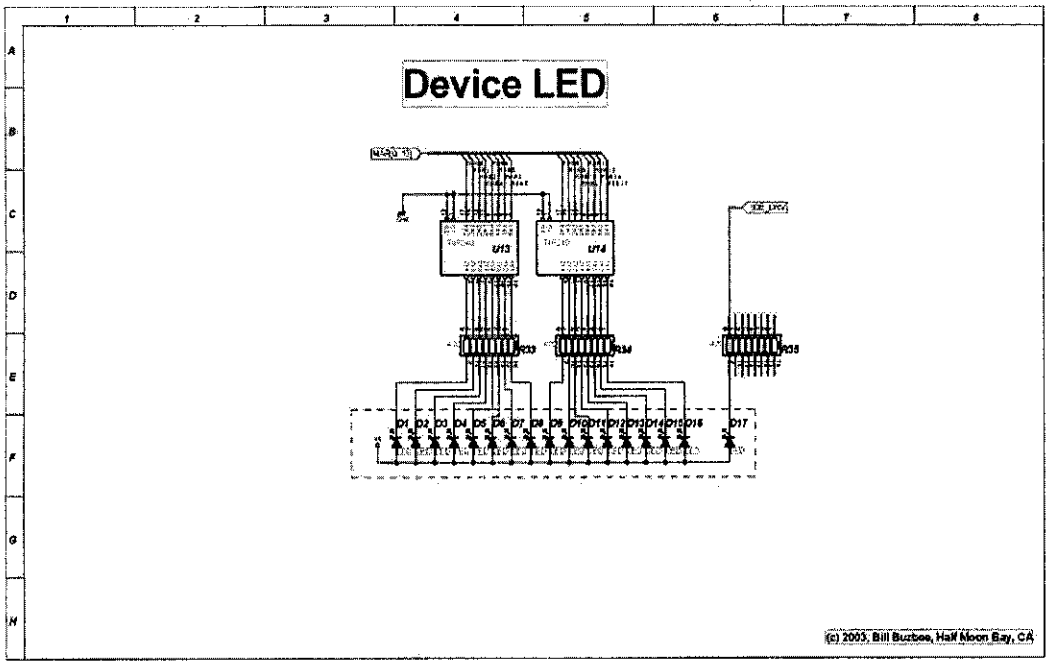

SRAM
A[0..15]
MEMORY
WR
U21
U22
U23
U24
U25
U26
U27
U28
U29
U30A
RW
IOCLK
DBUS[0..7]
(c) 2003, Bill Buzbee, Half Moon Bay, CA

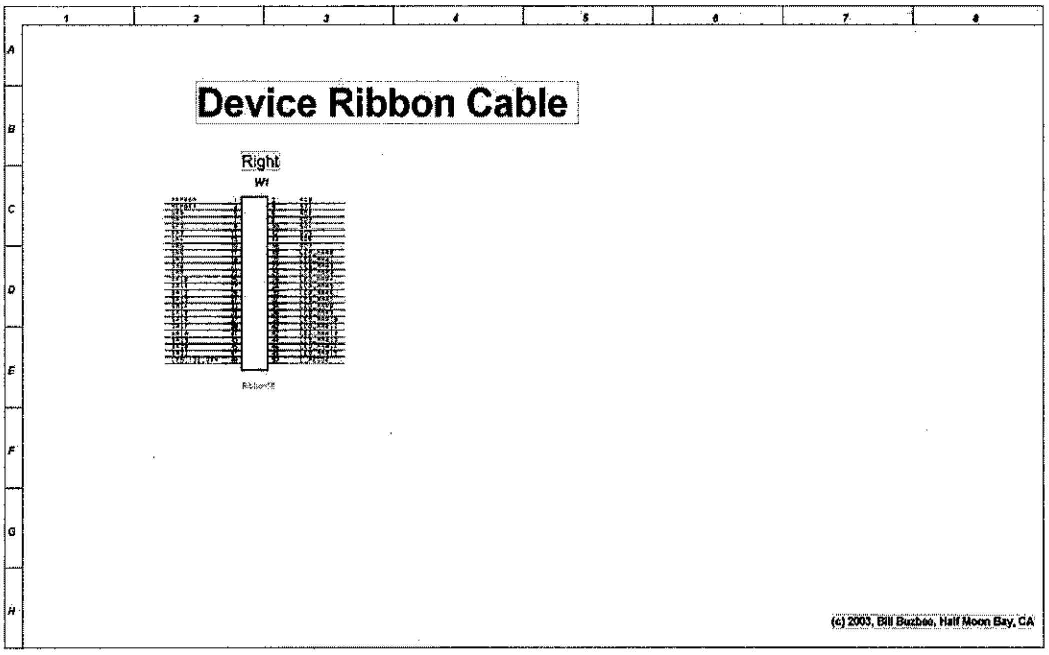

UARTS
A[0..15]
DBUS[0..7]
U26A
U26B
U27A
U17
U3C
MEMORY
U4D
U18
U27B
MEMREF
RESET
U7D
WR
PSEL0
PSEL1
PSEL2
PSEL3
PSEL4
PSEL5
PSEL6
PSEL7
Y1
1.8432MHz
U21
U15
16550
U7E
IRQ1
U7C
IRQ2
U22
U19
U20
MAX233
P2
P1
RW
IOCLK
U5D
OE
(c) 2003, Bill Buzbee, Half Moon Bay, CA

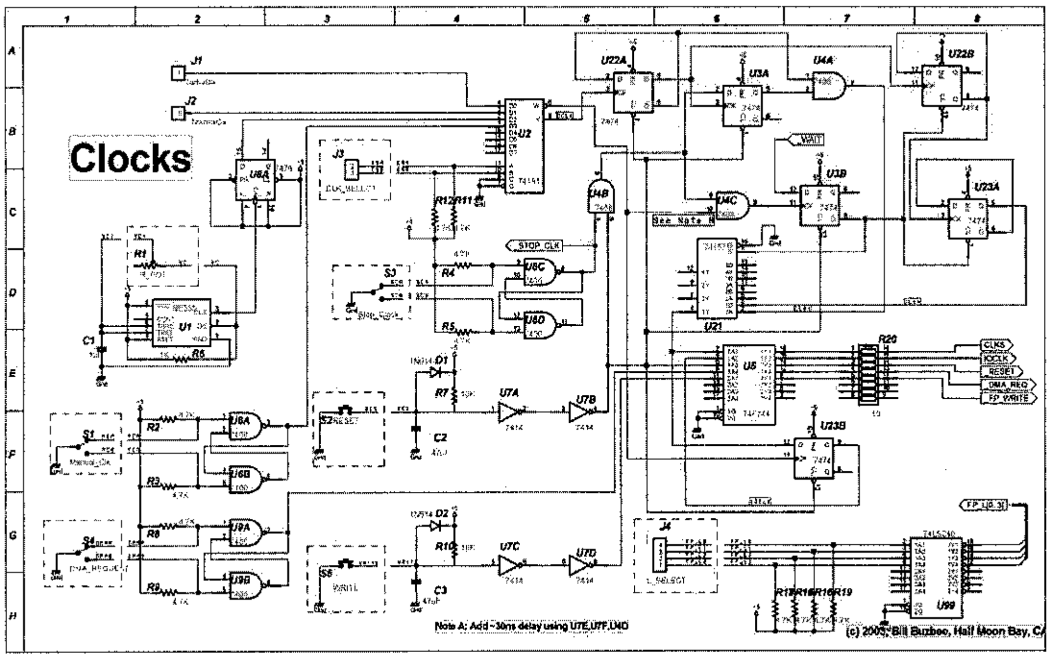

POST Display
IDE Interface
Real time clock
U27C
U5B
H1
U23
J1
U25
U24E
U7F
U24F
U7B
R99
C1
D20
D21
B1
L_PCODE
PSEL4
WR
DBUS[0..7]
A[0..15]
RESET
PSEL3
OE
IRQ3
PSEL9
IRQ1
IDE_DRV
1N5817
(c) 2003, Bill Buzbee, Half Moon Bay, CA


RAM & ROM
U2
U1
U8A
U8B
U6B
U7A
U6A
U3A
U3B
U4A
U4B
U28A
Currently unused...
U5C
U29
A[0..21]
DBUS[0..7]
OE
WR
RESET
PSEL1
FP_RAM
MEMORY
(c) 2003, Bill Buzbee, Half Moon Bay, CA

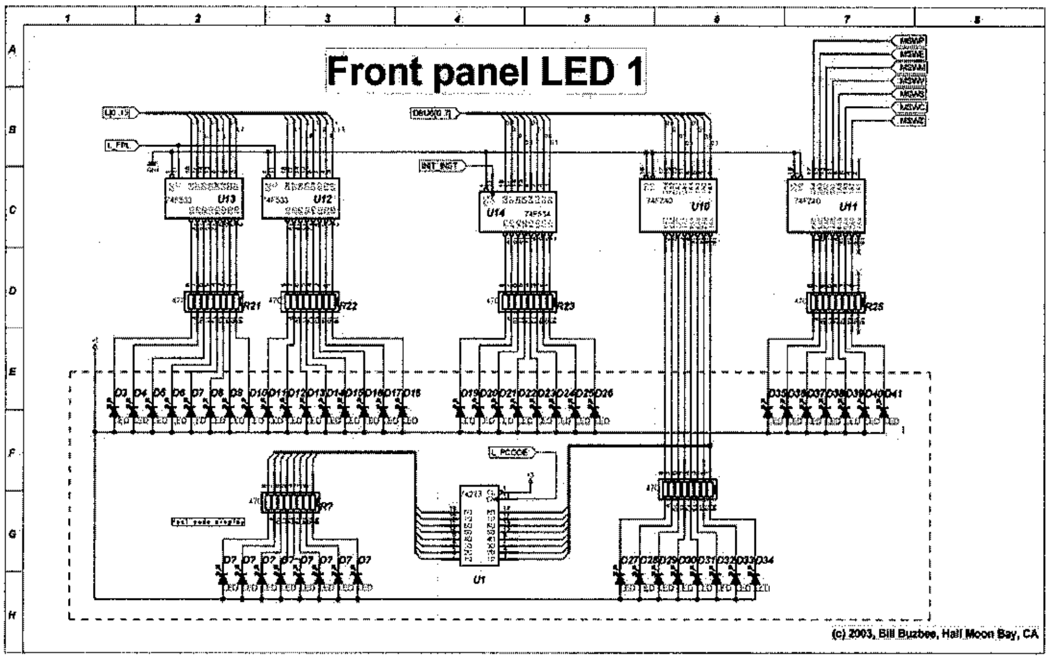

Device Card Switches
S31
S32
MEMORY/DEVICE
RAM/ROM
S23
S24
S25
S26
S27
S28
S29
S30
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
RW
DMA_ACK
U6C
U6D
7400
U4C
PSEL2
FP_RAM
U9
74F244
DBUS[0..15]
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
S12
S13
S14
S15
S16
S17
S18
S19
S20
S21
S22
R27
R28
R29
R30
R31
R32
R19
R20
R21
R22
R23
R24
R25
R26
R11
R12
R13
R14
R15
R16
R17
R18
U12
U11
U10
74F244
NOTE: SA21/A21 odd order reflects wiring mistake
MEMORY
A[0..21]
(c) 2003, Bill Buzbee, Half Moon Bay, CA

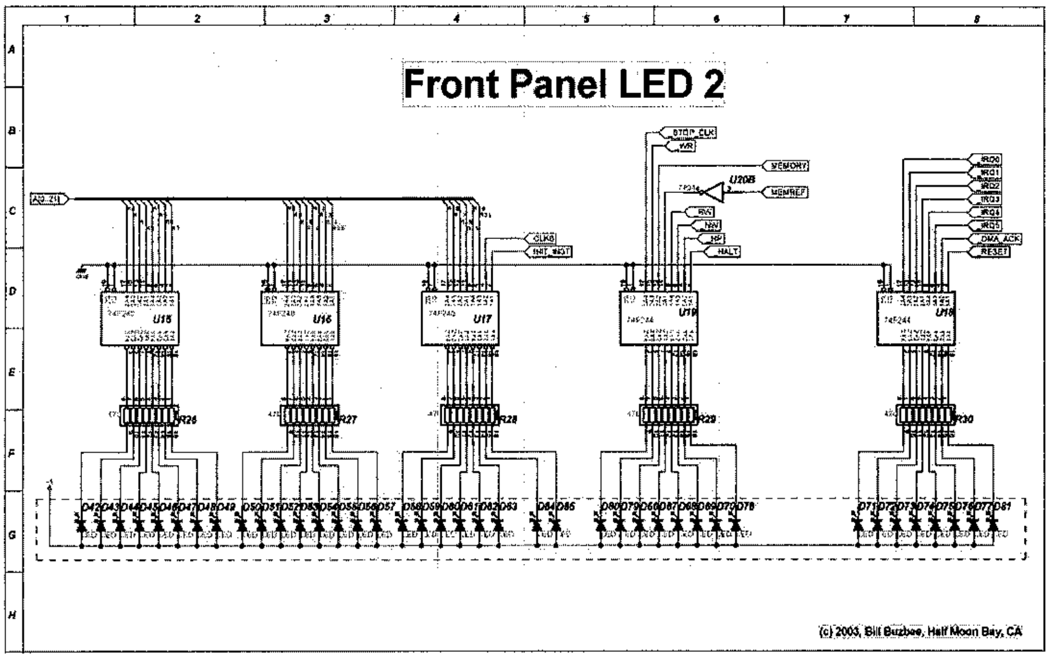

Device LED
MAR[0..15]
U13
U14
74F240
R33
R34
R35
D1
D2
D3
D4
D5
D6
D7
D8
D9
D10
D11
D12
D13
D14
D15
D16
D17
(c) 2003, Bill Buzbee, Half Moon Bay, CA

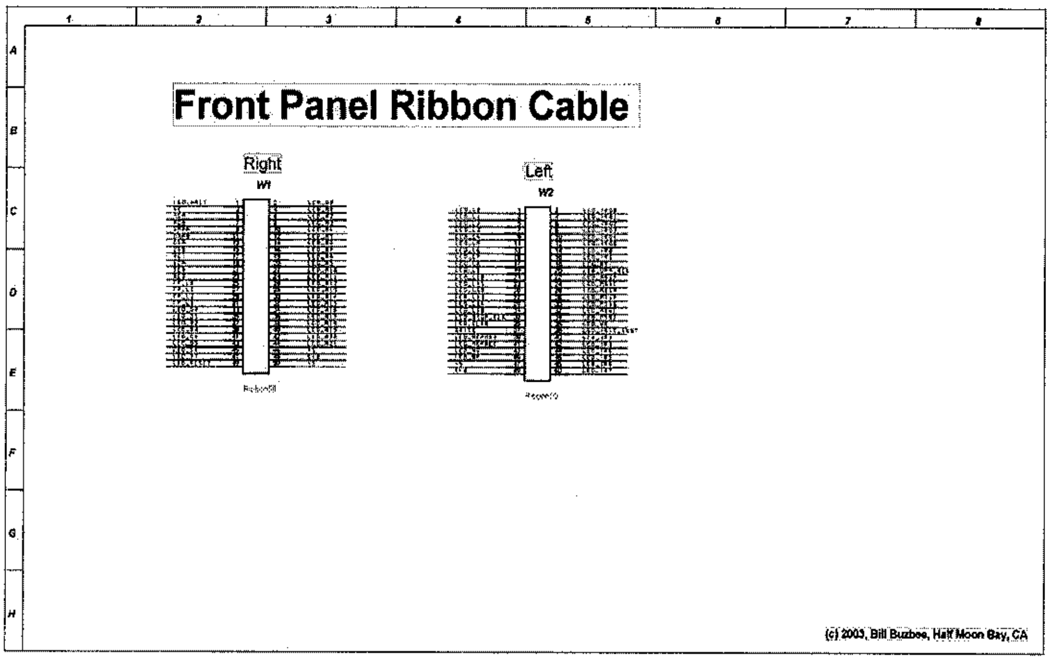

Device Ribbon Cable
Right
W1
Ribbon50
(c) 2003, Bill Buzbee, Half Moon Bay, CA


Clocks
J1
J2
J3
U2
U22A
U3A
U4A
U22B
U6A
U4B
U4C
U3B
U23A
R12 R11
STOP_CLK
WAIT
See Note A
R1
U1
C1
R6
S3
R4
U8C
U8D
R5
U21
U5
R20
CLKS
IOCLK
RESET
DMA_REQ
FP_WRITE
D1
R7
U7A
U7B
S2 RESET
C2
S1
R2
U8A
U8B
R3
U23B
S4
R8
U9A
U9B
R9
D2
R10
U7C
U7D
S5
WRITE
C3
J4
FP_IO[0..3]
U99
R17 R18 R16 R19
Note A: Add ~30ns delay using U7E,U7F,U4D
(c) 2003, Bill Buzbee, Half Moon Bay, CA

63


Front panel LED 1
U13
U12
U14
U10
U11
R21
R22
R23
R25
R7
U1
INIT_INGT
L_FPL
L_PCODE
(c) 2003, Bill Buzbee, Half Moon Bay, CA


Front Panel LED 2
U15
U16
U17
U19
U18
U20B
R26
R27
R28
R29
R30
STOP_CLK
WR
MEMORY
MEMREF
RW
NW
HP
HALT
CLKB
INIT_INGT
IRQ0
IRQ1
IRQ2
IRQ3
IRQ4
IRQ5
DMA_ACK
RESET
(c) 2003, Bill Buzbee, Half Moon Bay, CA


Front Panel Ribbon Cable
Right
W1
Left
W2
(c) 2003, Bill Buzbee, Half Moon Bay, CA

Appendix III – Microcode listing (extracted from www.homebrewcpu.com/working_microcode.htm)

System Microcode

This page is actually the source code for the M-1 microcode to be used in the microcode-level simulator, as well as the actual bits to be burned into the proms. It is processed by extracting the text and processing with cpp and a Perl script (see the Software page for more details). The created files are:

◆mcode.h - Description of the fields within the microinstruction word.
◆mcode.c - An initialized array representing the microcode image.
◆mcdefs.h - #defines for microcode fields.
◆prombits.h - The initialization declaration for the 512 56-bit microinstruction words.
◆prom0.hex .. prom4.hex - Hex images of the slices of the microcode store to be fed into the PROM programmer.
◆opcodes.h - Opcode strings.

```
//=====================================================
// BEGIN mcode.h
/* Define for micro instruction word.  Assume I'll be using 512x8 bipolar
 * PROMs.  This version is quite a bit more compact than previous ones,
 * but at the cost of having addition field decoding logic.  Initial plan
 * is to send these signals across the backplane and do decoding on the
 * appropriate card.
 *
 * Note that the encoding here is getting pretty ugly.  I'm trying hard to
 * keep the microcode store down to 5 PROMS - 16 bits for enable signals,
 * 16 bits for latch signals and 8 bits for the next field.
 */
typedef struct {
        unsigned next:8;  // Next micro-op to exec. 0x00 means
                          // use output of priority encoder, 0xff
                          // means use IR[0..7].  Also a significant
                          // bit of !(IR[0..7]==0xff) to give the full
                          // 9-bit microcode address.
        unsigned latch:4; // Register latch signal. Value:
                          // 0x0 : none
                          // 0x1 : MSW (flag nibble only, from Z)
                          // 0x2 : C
                          // 0x3 : PC
                          // 0x4 : DP
                          // 0x5 : SP
                          // 0x6 : A
                          // 0x7 : B
                          // 0x8 : MDR (from Z)
                          // 0x9 : PTB
                          // 0xa : [A low placeholder]
                          // 0xb : [A high placeholder]
                          // 0xc : [B low placeholder]
                          // 0xd : [C low placeholder]
                          // 0xe : [SSP placeholder]
                          // 0xf : IR_REG (IR[5..7])
        unsigned lmar:1;  // Latch MAR
        unsigned lmdrlo:1;        // Latch MDR(lo) from dbus
        unsigned lmdrhi:1;        // Latch MDR(hi) from dbus
        unsigned emdrlo:1;        // Drive dbus with MDR(lo)
        unsigned emdrhi:1;        // Drive dbus with MDR(hi)
        unsigned priv:1;          // Priviliged instruction
        unsigned lmode:1;         // Latch (M)ode bit in MSW
        unsigned lpaging:1;       // Latch (P)aging enable bit in MSW
        unsigned misc:4;  // Controls signals which never occur at the
                          // same time:
                          // 0x0 : none
                          // 0x1 :
                          // 0x2 : halt
                          // 0x3 :
                          // 0x4 : trap on overflow
                          // 0x5 : latch PTE
                          // 0x6 : set flags (from alu op)
                          // 0x7 : init_inst (clear MDR, PC->TPC, latch IR)
                          // 0x8 : right shift alu output
                          // 0x9 : DMA acknowledge
                          // 0xa : latch MSW[ie] (Interrupt Enable)
                          // 0xb : do branch
                          // 0xc : latch MSW[in_trap]
                          // 0xd : commit state
                          // 0xe :
                          // 0xf :
        unsigned e_l:4;            // Enable L bus
                                   // 0x0 : MAR
                                   // 0x1 : MSW
                                   // 0x2 : C
                                   // 0x3 : PC
                                   // 0x4 : DP
                                   // 0x5 : SP
                                   // 0x6 : A
                                   // 0x7 : B
                                   // 0x8 : MDR
                                   // 0x9 : PTB
                                   // 0xa : SSP
                                   // 0xb : TPC
                                   // 0xc :
                                   // 0xd :
                                   // 0xe :
                                   // 0xf : IR_BASE (4+IR[6..7])
        unsigned e_r:2;            // Enable R bus
                                   // 0x0 : MDR
                                   // 0x1 : Immediate
                                   // 0x2 : Fault code/encoder
                                   // 0x3 :
        unsigned immval:2;         // Immediate value
                                   // 0x0 : 0
                                   // 0x1 : 1
                                   // 0x2 : -2
                                   // 0x3 : -1
        unsigned aluop_size:1;     // 0x0 -> 16 bits, 0x1 -> 8 bits
        unsigned aluop:2;          // Which alu operation to perform
                                   // 0x0 : IR[1..3]
                                   // 0x1 : AND
                                   // 0x2 : SUB
                                   // 0x3 : ADD
        unsigned carry:1;          // 0x0 -> 0, 0x1 -> MSW[c]
        unsigned l_size:1;         // 0x0 -> latch byte, 0x1 -> latch word
        unsigned br_sense:1;       // 0x0 -> don't negate, 0x1 -> negate
                                   // Non-negated branch conditions are:
                                   //       0x0 : eq
                                   //       0x1 : eq
                                   //       0x2 : lt
                                   //       0x3 : le
                                   //       0x4 : ltu
                                   //       0x5 : leu
                                   //       0x6 : eq
                                   //       0x7 : ne
        unsigned user_ptb:1;       // User page table base override
        unsigned code_ptb:1;// 0 to select data region of PTB, 1 for code
} mcode_rec_t;

extern mcode_rec_t mcode_store[512];
```

```
// END mcode.h

//=======================================================
// BEGIN mcode.c
//
#include "mcode.h"
mcode_rec_t mcode_store[512] = {
#include "prombits.h"
};

// END mcode.c

//=======================================================
// PREPROCESS prombits.h
// BEGIN mcdefs.h

// Register defines for LATCH() and EL()
#define  R_MSW    1
#define  R_C      2
#define  R_PC     3
#define  R_DP     4
#define  R_SP     5
#define  R_A      6
#define  R_B      7
#define  R_MDR    8
#define  R_PTB    9
#define  R_SSP    10

// Register defines for LATCH()-only
#define  R_NONE   0
#define  R_IR_REG 15

// Register defines for EL()-only
#define  R_MAR    0
#define  R_TPC    11
#define R_FCODE   12
#define  R_IR_BASE         15

// Register defines for ER()
#define  ER_MDR   0
#define  ER_IMM   1
#define  ER_FAULT 2
// Defines for IMMVAL()
#define  IMM_0    0
#define  IMM_1    1
#define  IMM_NEG1 3
#define  IMM_NEG2 2

// Defines for MISC()
#define  M_NONE   0
#define M_SYSCALL 1
#define  M_HALT   2
#define M_BKPT 3
#define  M_TRAPO  4
#define  M_LPTE   5
#define  M_SET_FLAGS       6
#define  M_INIT_INST       7
#define  M_RSHIFT 8
#define  M_DMA_ACK         9
#define  M_LEI    10
#define  M_DO_BRANCH       11
#define M_CLR_TRAP         12
#define M_COMMIT 13

// Defines for ALUOP(op,size,carry)
#define  OP_IR13  0
#define  OP_AND   1
#define  OP_SUB   2
```

```
#define  OP_ADD   3
#define  WORD     0
#define  BYTE     1
#define  LWORD    1
#define  LBYTE    0
#define  NO_CARRY 0
#define  CARRY_IN 1

// Defines for CBR()
#define  B_NORMAL 0
#define  B_NEGATED        1

// END mcdefs.h

#define NEXT_POS   0
#define LATCH_POS  1
#define LMAR_POS   2
#define LMDRLO_POS         3
#define LMDRHI_POS         4
#define EMDRLO_POS         5
#define EMDRHI_POS         6
#define PRIV_POS   7
#define LMODE_POS  8
#define LPAGING_POS        9
#define MISC_POS   10
#define E_L_POS            11
#define E_R_POS            12
#define IMMVAL_POS         13
#define ALUOP_SIZE_POS     14
#define ALUOP_POS  15
#define CARRY_POS  16
#define L_SIZE_POS         17
#define BR_SENSE_POS       18
#define USER_PTB_POS       19
#define CODE_PTB_POS       20
#define NEXT(VAL) INIT(NEXT_POS,VAL)
#define LATCH(VAL)         INIT(LATCH_POS,VAL)
#define LMAR(VAL) INIT(LMAR_POS,VAL)
#define LMDRLO(VAL)        INIT(LMDRLO_POS,VAL)
#define LMDRHI(VAL)        INIT(LMDRHI_POS,VAL)
#define EMDRLO(VAL)        INIT(EMDRLO_POS,VAL)
#define EMDRHI(VAL)        INIT(EMDRHI_POS,VAL)
#define PRIV(VAL) INIT(PRIV_POS,VAL)
#define LMODE(VAL)         INIT(LMODE_POS,VAL)
#define LPAGING(VAL)       INIT(LPAGING_POS,VAL)
#define MISC(VAL) INIT(MISC_POS,VAL)
#define E_L(VAL)  INIT(E_L_POS,VAL)
#define E_R(VAL)  INIT(E_R_POS,VAL)
#define IMMVAL(VAL)        INIT(IMMVAL_POS,VAL)
#define ALUOP_SIZE(VAL)    INIT(ALUOP_SIZE_POS,VAL)
#define ALUOP(VAL)         INIT(ALUOP_POS,VAL)
#define CARRY(VAL)         INIT(CARRY_POS,VAL)
#define L_SIZE(VAL)        INIT(L_SIZE_POS,VAL)
#define BR_SENSE(VAL)      INIT(BR_SENSE_POS,VAL)
#define USER_PTB(VAL)      INIT(USER_PTB_POS,VAL)
#define CODE_PTB(VAL)      INIT(CODE_PTB_POS,VAL)
#define CBR(SENSE,TGT)     MISC(M_DO_BRANCH),BR_SENSE(SENSE),NEXT(TGT)
#define L(REG,SIZE)        LATCH(REG),L_SIZE(SIZE)
#define  USE_IR   0xff
#define READLO LMDRLO(1)
#define READHI LMDRHI(1)
#define READEXT LMDRLO(1),LMDRHI(1)
#define WRITELO EMDRLO(1)
#define WRITEHI EMDRHI(1)
#define INC_TO_Z(REG)
E_L(REG),E_R(ER_IMM),IMMVAL(IMM_1),ALU(OP_ADD,WORD,NO_CARRY)
#define INC2_TO_Z(REG)
E_L(REG),E_R(ER_IMM),IMMVAL(IMM_NEG2),ALU(OP_SUB,WORD,NO_CARRY)
```

```
#define DEC_TO_Z(REG)
E_L(REG),E_R(ER_IMM),IMMVAL(IMM_NEG1),ALU(OP_ADD,WORD,NO_CARRY)
#define ZERO_TO_Z
E_L(R_MDR),E_R(ER_IMM),IMMVAL(IMM_0),ALU(OP_AND,WORD,NO_CARRY)
#define NEG1_TO_Z
E_L(R_MDR),E_R(ER_IMM),IMMVAL(IMM_NEG1),ALU(OP_ADD,WORD,NO_CARRY)
#define TO_Z(REG)
E_L(REG),E_R(ER_IMM),IMMVAL(IMM_NEG1),ALU(OP_AND,WORD,NO_CARRY)
#define TO_Z8(REG)
E_L(REG),E_R(ER_IMM),IMMVAL(IMM_NEG1),ALU(OP_AND,BYTE,NO_CARRY)
#define LDHI READHI,INC_TO_Z(R_MAR),LMAR(1)
#define LDLO READLO,TO_Z(R_PC),LMAR(1)
#define STHI WRITEHI,INC_TO_Z(R_MAR),LMAR(1)
#define STLO WRITELO,TO_Z(R_PC),LMAR(1)
#define LDIMMHI CODE_PTB(1),READHI,L(R_PC,LWORD),INC_TO_Z(R_PC),LMAR(1)
#define LDIMMLO CODE_PTB(1),READLO,L(R_PC,LWORD),INC_TO_Z(R_PC),LMAR(1)
#define LDIMMEXT CODE_PTB(1),READEXT,L(R_PC,LWORD),INC_TO_Z(R_PC),LMAR(1)
#define GEN_ADDR(BASE) E_L(BASE),E_R(ER_MDR),ALU(OP_ADD,WORD,NO_CARRY)
#define COMPARE_0(REG)
E_L(REG),E_R(ER_IMM),IMMVAL(IMM_0),ALU(OP_SUB,WORD,NO_CARRY)
#define COMPARE8_0(REG)
E_L(REG),E_R(ER_IMM),IMMVAL(IMM_0),ALU(OP_SUB,BYTE,NO_CARRY)
#define ALU(OP,SZ,CRY) ALUOP(OP),ALUOP_SIZE(SZ),CARRY(CRY)
#define FETCH_OP
CODE_PTB(1),READLO,MISC(M_INIT_INST),INC_TO_Z(R_MAR),L(R_PC,LWORD),LMAR(1),N
EXT(UNUSABLE)
#define PUSHLO WRITELO,DEC_TO_Z(R_MAR),LMAR(1)
#define PUSHHI WRITEHI,DEC_TO_Z(R_MAR),LMAR(1)
#define POPLO READLO,INC_TO_Z(R_MAR),LMAR(1)
#define POPHI READHI,INC_TO_Z(R_MAR),LMAR(1)
#define TO_MDR(REG) TO_Z(REG),L(R_MDR,LWORD)
#define FROM_MDR(REG) TO_Z(R_MDR),L(REG,LWORD)
```

Bottom half of PROM - (starting point of each instruction, using opcode as direct index)

0x00	halt	;	MISC(M_HALT),DEC_TO_Z(R_PC),L(R_PC,LWORD),LMAR(1),NEXT(Fetch)
0x01	ld.8 A,#u16_u8_10(SP)	;	LDIMMHI,NEXT(Lda8_16)
0x02	push C	;	TO_Z(R_C),L(R_MDR,LWORD),NEXT(Push16)
0x03	push PC	;	TO_Z(R_TPC),L(R_MDR,LWORD),NEXT(Push16)
0x04	push DP	;	TO_Z(R_DP),L(R_MDR,LWORD),NEXT(Push16)
0x05	ld.8 B,#u16_u8_10(SP)	;	LDIMMHI,NEXT(Ldb8_16)
0x06	push A	;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(Push16)
0x07	push B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Push16)
0x08	br.ne #d16	;	LDIMMHI,NEXT(BrNegated)
0x09	pop MSW	;	TO_Z(R_SP),LMAR(1),NEXT(Pop16)
0x0a	pop C	;	TO_Z(R_SP),LMAR(1),NEXT(Pop16)
0x0b	pop PC	;	TO_Z(R_SP),LMAR(1),NEXT(Pop16)
0x0c	pop DP	;	TO_Z(R_SP),LMAR(1),NEXT(Pop16)

0x0d	pop SP	;	TO_Z(R_SP),LMAR(1),NEXT(Pop16)
0x0e	pop A	;	TO_Z(R_SP),LMAR(1),NEXT(Pop16)
0x0f	pop B	;	TO_Z(R_SP),LMAR(1),NEXT(Pop16)
0x10	ld.8 A,#u16(DP)	;	LDIMMHI,NEXT(Lda8_16)
0x11	ld.8 A,#u8(SP)	;	LDIMMLO,NEXT(Lda8_8)
0x12	ld.8 A,#u8(A)	;	LDIMMLO,NEXT(Lda8_8)
0x13	ld.8 A,#u8(B)	;	LDIMMLO,NEXT(Lda8_8)
0x14	ld.8 B,#u16(DP)	;	LDIMMHI,NEXT(Ldb8_16)
0x15	ld.8 B,#u8(SP)	;	LDIMMLO,NEXT(Ldb8_8)
0x16	ld.8 B,#u8(A)	;	LDIMMLO,NEXT(Ldb8_8)
0x17	ld.8 B,#u8(B)	;	LDIMMLO,NEXT(Ldb8_8)
0x18	ld.16 A,#u16(DP)	;	LDIMMHI,NEXT(Lda16_16)
0x19	ld.16 A,#u16_u8_68(SP)	;	LDIMMHI,NEXT(Lda16_16)
0x1a	ld.16 A,#u8(A)	;	LDIMMLO,NEXT(Lda16_8)
0x1b	ld.16 A,#u8(B)	;	LDIMMLO,NEXT(Lda16_8)
0x1c	ld.16 B,#u16(DP)	;	LDIMMHI,NEXT(Ldb16_16)
0x1d	ld.16 B,#u16_u8_68(SP)	;	LDIMMHI,NEXT(Ldb16_16)
0x1e	ld.16 B,#u8(A)	;	LDIMMLO,NEXT(Ldb16_8)
0x1f	ld.16 B,#u8(B)	;	LDIMMLO,NEXT(Ldb16_8)
0x20	sub.8 A,#u16(DP)	;	LDIMMHI,NEXT(Aluop8_indir16)
0x21	sub.8 A,#u8(SP)	;	LDIMMLO,NEXT(Aluop8_indir)
0x22	push MSW	;	TO_Z(R_MSW),L(R_MDR,LWORD),NEXT(Push16)
0x23	sub.8 A,#u8(B)	;	LDIMMLO,NEXT(Aluop8_indir)
0x24	sub.8 A,#i8_1	;	LDIMMLO,NEXT(Aluop8)
0x25	sub.8 A,#1	;	INC_TO_Z(R_MDR),L(R_MDR,LWORD),NEXT(Aluop8)

0x26	push SP	;	TO_Z(R_SP),L(R_MDR,LWORD),NEXT(Push16)
0x27	sub.8 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Aluop8)
0x28	sub.16 A,#u16(DP)	;	LDIMMHI,NEXT(Aluop16 indir16)
0x29	sub.16 A,#u8(SP)	;	LDIMMLO,NEXT(Aluop16 indir)
0x2a	nop0	;	NEXT(Fetch)
0x2b	sub.16 A,#u8(B)	;	LDIMMLO,NEXT(Aluop16 indir)
0x2c	sub.16 A,#i16_exti8	;	LDIMMHI,NEXT(Aluop16 16)
0x2d	sub.16 A,#exti8	;	LDIMMEXT,NEXT(Aluop16)
0x2e	wcpte A,(B)	;	PRIV(1),TO_Z(R_B),LMAR(1),NEXT(Wcpte)
0x2f	sub.16 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Aluop16)
0x30	add.8 A,#u16(DP)	;	LDIMMHI,NEXT(Aluop8 indir16)
0x31	add.8 A,#u8(SP)	;	LDIMMLO,NEXT(Aluop8 indir)
0x32	br A	;	TO_Z(R_A),L(R_PC,LWORD),LMAR(1),NEXT(Fetch)
0x33	add.8 A,#u8(B)	;	LDIMMLO,NEXT(Aluop8 indir)
0x34	add.8 A,#i8_1	;	LDIMMLO,NEXT(Aluop8)
0x35	add.8 A,#1	;	INC_TO_Z(R_MDR),L(R_MDR,LWORD),NEXT(Aluop8)
0x36	add.8 A,A	;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(Aluop8)
0x37	add.8 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Aluop8)
0x38	add.16 A,#u16(DP)	;	LDIMMHI,NEXT(Aluop16 indir16)
0x39	add.16 A,#u8(SP)	;	LDIMMLO,NEXT(Aluop16 indir)
0x3a	syscall #sys_num8	;	LDIMMLO,NEXT(Syscall)
0x3b	add.16 A,#u8(B)	;	LDIMMLO,NEXT(Aluop16 indir)
0x3c	add.16 A,#i16_exti8	;	LDIMMHI,NEXT(Aluop16 16)
0x3d	add.16 A,#exti8	;	LDIMMEXT,NEXT(Aluop16)
0x3e	add.16 A,A	;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(Aluop16)
0x3f	add.16 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Aluop16)

0x40	cmp.8 A,#u16(DP)	;	LDIMMHI,NEXT(Cmp8_indir16)
0x41	cmp.8 A,#u8(SP)	;	LDIMMLO,NEXT(Cmp8_indir)
0x42	copy C,B	;	TO_Z(R_B),L(R_C,LWORD),NEXT(Fetch)
0x43	cmp.8 A,#u8(B)	;	LDIMMLO,NEXT(Cmp8_indir)
0x44	cmp.8 A,#i8_0	;	LDIMMLO,NEXT(Cmp8)
0x45	cmp.8 A,#0	;	E_L(R_A),E_R(ER_MDR),ALU(OP_SUB,BYTE,NO_CARRY),MISC(M_SET_FLAGS),NEXT(Fetch)
0x46	xor.16 A,A	;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(Aluop16)
0x47	cmp.8 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Cmp8)
0x48	cmp.16 A,#u16(DP)	;	LDIMMHI,NEXT(Cmp16_indir16)
0x49	cmp.16 A,#u8(SP)	;	LDIMMLO,NEXT(Cmp16_indir)
0x4a	sh0add B,A,B	;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(LeaB1)
0x4b	cmp.16 A,#u8(B)	;	LDIMMLO,NEXT(Cmp16_indir)
0x4c	cmp.16 A,#i16_exti8_0	;	LDIMMHI,NEXT(Cmp16_16)
0x4d	cmp.16 A,#exti8_0	;	LDIMMEXT,NEXT(Cmp16)
0x4e	cmp.16 A,#0	;	E_L(R_A),E_R(ER_MDR),ALU(OP_SUB,WORD,NO_CARRY),MISC(M_SET_FLAGS),NEXT(Fetch)
0x4f	cmp.16 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Cmp16)
0x50	or.8 A,#u16(DP)	;	LDIMMHI,NEXT(Aluop8_indir16)
0x51	or.8 A,#u8(SP)	;	LDIMMLO,NEXT(Aluop8_indir)
0x52	sex A	;	TO_Z8(R_A),L(R_A,LWORD),NEXT(Fetch)
0x53	or.8 A,#u8(B)	;	LDIMMLO,NEXT(Aluop8_indir)
0x54	or.8 A,#i8_1	;	LDIMMLO,NEXT(Aluop8)
0x55	or.8 A,#1	;	INC_TO_Z(R_MDR),L(R_MDR,LWORD),NEXT(Aluop8)
0x56	br.leu #d16	;	LDIMMHI,NEXT(BrNormal)
0x57	or.8 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Aluop8)
0x58	or.16 A,#u16(DP)	;	LDIMMHI,NEXT(Aluop16_indir16)

0x59	or.16 A,#u8(SP)	;	LDIMMLO,NEXT(Aluop16_indir)
0x5a	shladd A,B,A	;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(LeaABA2)
0x5b	or.16 A,#u8(B)	;	LDIMMLO,NEXT(Aluop16_indir)
0x5c	or.16 A,#i16_exti8	;	LDIMMHI,NEXT(Aluop16_16)
0x5d	or.16 A,#exti8	;	LDIMMEXT,NEXT(Aluop16)
0x5e	br.gtu #d16	;	LDIMMHI,NEXT(BrNegated)
0x5f	or.16 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Aluop16)
0x60	and.8 A,#u16(DP)	;	LDIMMHI,NEXT(Aluop8_indir16)
0x61	and.8 A,#u8(SP)	;	LDIMMLO,NEXT(Aluop8_indir)
0x62	shladd B,A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(LeaBAB2)
0x63	and.8 A,#u8(B)	;	LDIMMLO,NEXT(Aluop8_indir)
0x64	and.8 A,#i8_1	;	LDIMMLO,NEXT(Aluop8)
0x65	and.8 A,#1	;	INC_TO_Z(R_MDR),L(R_MDR,LWORD),NEXT(Aluop8)
0x66	nop1	;	NEXT(Fetch)
0x67	and.8 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Aluop8)
0x68	and.16 A,#u16(DP)	;	LDIMMHI,NEXT(Aluop16_indir16)
0x69	and.16 A,#u8(SP)	;	LDIMMLO,NEXT(Aluop16_indir)
0x6a	shladd B,B,A	;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(LeaBBA2)
0x6b	and.16 A,#u8(B)	;	LDIMMLO,NEXT(Aluop16_indir)
0x6c	and.16 A,#i16_exti8	;	LDIMMHI,NEXT(Aluop16_16)
0x6d	and.16 A,#exti8	;	LDIMMEXT,NEXT(Aluop16)
0x6e	strcopy	;	TO_Z(R_B),LMAR(1),NEXT(Strcopy)
0x6f	and.16 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Aluop16)
0x70	lea A,#u16(DP)	;	LDIMMHI,NEXT(LdaA_16)
0x71	lea A,#u16(SP)	;	LDIMMHI,NEXT(LdaA_16)
0x72	lea A,#u16(A)	;	LDIMMHI,NEXT(LdaA_16)

0x73	lea A,#u16(B)	;	LDIMMHI,NEXT(LdaA_16)
0x74	lea B,#u16(DP)	;	LDIMMHI,NEXT(LdaB_16)
0x75	lea B,#u16(SP)	;	LDIMMHI,NEXT(LdaB_16)
0x76	lea B,#u16(A)	;	LDIMMHI,NEXT(LdaB_16)
0x77	lea B,#u16(B)	;	LDIMMHI,NEXT(LdaB_16)
0x78	ld.8 A,#u8	;	LDIMMLO,NEXT(LdiA8)
0x79	ld.8 B,#u8	;	LDIMMLO,NEXT(LdiB8)
0x7a	ld.16 A,#exti8_u16	;	LDIMMEXT,NEXT(LdiA16)
0x7b	ld.16 B,#exti8_u16	;	LDIMMEXT,NEXT(LdiB16)
0x7c	ld.16 A,#u16	;	LDIMMHI,NEXT(LdiA16_lo)
0x7d	ld.16 B,#u16	;	LDIMMHI,NEXT(LdiB16_lo)
0x7e	adc.16 A,A	;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(Adc16)
0x7f	adc.16 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Adc16)
0x80	call #d16	;	INC2_TO_Z(R_PC),L(R_MDR,LWORD),NEXT(CallImm)
0x81	ld.16 A,#u8(SP)	;	LDIMMLO,NEXT(Lda16_8)
0x82	call A	;	TO_Z(R_PC),L(R_MDR,LWORD),NEXT(CallA)
0x83	br #d16_d8	;	LDIMMHI,NEXT(RelBrLo)
0x84	sbr #d8	;	LDIMMEXT,NEXT(RelBr)
0x85	ld.16 B,#u8(SP)	;	LDIMMLO,NEXT(Ldb16_8)
0x86	lea A,#u16(PC)	;	LDIMMHI,NEXT(LeaPC)
0x87	lea B,#u16(PC)	;	LDIMMHI,NEXT(LeaPC)
0x88	copy A,MSW	;	TO_Z(R_MSW),L(R_A,LWORD),NEXT(Fetch)
0x89	br.eq #d16	;	LDIMMHI,NEXT(BrNormal)
0x8a	reti	;	PRIV(1),NEXT(Reti)
0x8b	trapo	;	MISC(M_TRAPO),NEXT(Fetch)
0x8c	bset.8 A,#mask8,#d8	;	LDIMMLO,NEXT(Bset8)
0x8d	bclr.8 A,#mask8,#d8	;	LDIMMLO,NEXT(Bclr8)

0x8e	bset.16 A,#mask16,#d8	;	LDIMMHI,NEXT(Bset16)
0x8f	bclr.16 A,#mask16,#d8	;	LDIMMHI,NEXT(Bclr16)
0x90	cmpb.eq.8 A,#u16(DP),#d8	;	LDIMMHI,NEXT(Cmpb8_indir16)
0x91	cmpb.eq.8 A,#u8(SP),#d8	;	LDIMMLO,NEXT(Cmpb8_indir)
0x92	copy B,A	;	TO_Z(R_A),L(R_B,LWORD),NEXT(Fetch)
0x93	cmpb.eq.8 A,#u8(B),#d8	;	LDIMMLO,NEXT(Cmpb8_indir)
0x94	cmpb.eq.8 A,#i8_0,#d8	;	LDIMMLO,NEXT(Cmpb8)
0x95	cmpb.eq.8 A,#0,#d8	;	TO_Z(R_MDR),L(R_MDR,LWORD),NEXT(Cmpb8)
0x96	copy C,A	;	TO_Z(R_A),L(R_C,LWORD),NEXT(Fetch)
0x97	cmpb.eq.8 A,B,#d8	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Cmpb8)
0x98	cmpb.eq.16 A,#u16(DP),#d8	;	LDIMMHI,NEXT(Cmpb16_indir16)
0x99	cmpb.eq.16 A,#u8(SP),#d8	;	LDIMMLO,NEXT(Cmpb16_indir)
0x9a	copy A,B	;	TO_Z(R_B),L(R_A,LWORD),NEXT(Fetch)
0x9b	cmpb.eq.16 A,#u8(B),#d8	;	LDIMMLO,NEXT(Cmpb16_indir)
0x9c	cmpb.eq.16 A,#i16_exti8_0,#d8	;	LDIMMHI,NEXT(Cmpb16_16)
0x9d	cmpb.eq.16 A,#exti8_0,#d8	;	LDIMMEXT,NEXT(Cmpb16)
0x9e	cmpb.eq.16 A,#0,#d8	;	TO_Z(R_MDR),L(R_MDR,LWORD),NEXT(Cmpb16)
0x9f	cmpb.eq.16 A,B,#d8	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Cmpb16)
0xa0	cmpb.lt.8 A,#u16(DP),#d8	;	LDIMMHI,NEXT(Cmpb8_indir16)
0xa1	cmpb.lt.8 A,#u8(SP),#d8	;	LDIMMLO,NEXT(Cmpb8_indir)
0xa2	sh0add A,A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(LeaA1)
0xa3	cmpb.lt.8 A,#u8(B),#d8	;	LDIMMLO,NEXT(Cmpb8_indir)

0xa4	cmpb.lt.8 A,#i8_0,#d8	;	LDIMMLO,NEXT(Cmpb8)
0xa5	cmpb.lt.8 A,#0,#d8	;	TO_Z(R_MDR),L(R_MDR,LWORD),NEXT(Cmpb8)
0xa6	br.lt #d16	;	LDIMMHI,NEXT(BrNormal)
0xa7	cmpb.lt.8 A,B,#d8	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Cmpb8)
0xa8	cmpb.lt.16 A,#u16(DP),#d8	;	LDIMMHI,NEXT(Cmpb16_indir16)
0xa9	cmpb.lt.16 A,#u8(SP),#d8	;	LDIMMLO,NEXT(Cmpb16_indir)
0xaa	shladd A,A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(LeaAAB2)
0xab	cmpb.lt.16 A,#u8(B),#d8	;	LDIMMLO,NEXT(Cmpb16_indir)
0xac	cmpb.lt.16 A,#i16_exti8,#d8	;	LDIMMHI,NEXT(Cmpb16_16)
0xad	cmpb.lt.16 A,#exti8,#d8	;	LDIMMEXT,NEXT(Cmpb16)
0xae	br.ge #d16	;	LDIMMHI,NEXT(BrNegated)
0xaf	cmpb.lt.16 A,B,#d8	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Cmpb16)
0xb0	cmpb.le.8 A,#u16(DP),#d8	;	LDIMMHI,NEXT(Cmpb8_indir16)
0xb1	cmpb.le.8 A,#u8(SP),#d8	;	LDIMMLO,NEXT(Cmpb8_indir)
0xb2	sex B	;	TO_Z8(R_B),L(R_B,LWORD),NEXT(Fetch)
0xb3	cmpb.le.8 A,#u8(B),#d8	;	LDIMMLO,NEXT(Cmpb8_indir)
0xb4	cmpb.le.8 A,#i8,#d8	;	LDIMMLO,NEXT(Cmpb8)
0xb5	br.le #d16	;	LDIMMHI,NEXT(BrNormal)
0xb6	copy DP,A	;	TO_Z(R_A),L(R_DP,LWORD),NEXT(Fetch)
0xb7	cmpb.le.8 A,B,#d8	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Cmpb8)
0xb8	cmpb.le.16 A,#u16(DP),#d8	;	LDIMMHI,NEXT(Cmpb16_indir16)
0xb9	cmpb.le.16 A,#u8(SP),#d8	;	LDIMMLO,NEXT(Cmpb16_indir)
0xba	copy A,DP	;	TO_Z(R_DP),L(R_A,LWORD),NEXT(Fetch)

0xbb	cmpb.le.16 A,#u8(B),#d8	;	LDIMMLO,NEXT(Cmpb16_indir)
0xbc	cmpb.le.16 A,#i16_exti8, #d8	;	LDIMMHI,NEXT(Cmpb16_16)
0xbd	cmpb.le.16 A,#exti8,#d8	;	LDIMMEXT,NEXT(Cmpb16)
0xbe	br.gt #d16	;	LDIMMHI,NEXT(BrNegated)
0xbf	cmpb.le.16 A,B,#d8	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Cmpb16)
0xc0	br.geu #d16	;	LDIMMHI,NEXT(BrNegated)
0xc1	st.8 #u16_u8_10(SP),A	;	LDIMMHI,NEXT(Sta8_16)
0xc2	shl.16 A	;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(Shla16)
0xc3	shr.16 A	;	TO_Z(R_A),MISC(M_RSHIFT),L(R_A,LWORD),NEXT(Fetch)
0xc4	shl.16 B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Shlb16)
0xc5	st.8 #u16_u8_10(SP),B	;	LDIMMHI,NEXT(Stb8_16)
0xc6	shr.16 B	;	TO_Z(R_B),MISC(M_RSHIFT),L(R_B,LWORD),NEXT(Fetch)
0xc7	xor.16 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Aluop16)
0xc8	copy PTB,A	;	PRIV(1),TO_Z(R_A),L(R_PTB,LWORD),NEXT(Fetch)
0xc9	st.16 #u16_u8_10(SP),A	;	LDIMMHI,NEXT(Sta16_16)
0xca	copy MSW,A	;	PRIV(1),NEXT(CopyMSWA)
0xcb	copy SP,A	;	TO_Z(R_A),L(R_SP,LWORD),NEXT(Fetch)
0xcc	ld.16 C,#exti8_u16	;	LDIMMEXT,NEXT(LdiC16)
0xcd	st.16 #u16_u8_10(SP),B	;	LDIMMHI,NEXT(Stb16_16)
0xce	ld.16 C,#u16	;	LDIMMHI,NEXT(LdiC16_lo)
0xcf	br.ltu #d16	;	LDIMMHI,NEXT(BrNormal)
0xd0	st.8 #u16(DP),A	;	LDIMMHI,NEXT(Sta8_16)
0xd1	st.8 #u8(SP),A	;	LDIMMLO,NEXT(Sta8_8)
0xd2	st.8 #u8(A),A	;	LDIMMLO,NEXT(Sta8_8)
0xd3	st.8 #u8(B),A	;	LDIMMLO,NEXT(Sta8_8)

0xd4	st.8 #u16(DP),B	;	LDIMMHI,NEXT(Stb8_16)
0xd5	st.8 #u8(SP),B	;	LDIMMLO,NEXT(Stb8_8)
0xd6	st.8 #u8(A),B	;	LDIMMLO,NEXT(Stb8_8)
0xd7	st.8 #u8(B),B	;	LDIMMLO,NEXT(Stb8_8)
0xd8	st.16 #u16(DP),A	;	LDIMMHI,NEXT(Sta16_16)
0xd9	st.16 #u8(SP),A	;	LDIMMLO,NEXT(Sta16_8)
0xda	st.16 #u8(A),A	;	LDIMMLO,NEXT(Sta16_8)
0xdb	st.16 #u8(B),A	;	LDIMMLO,NEXT(Sta16_8)
0xdc	st.16 #u16(DP),B	;	LDIMMHI,NEXT(Stb16_16)
0xdd	st.16 #u8(SP),B	;	LDIMMLO,NEXT(Stb16_8)
0xde	st.16 #u8(A),B	;	LDIMMLO,NEXT(Stb16_8)
0xdf	st.16 #u8(B),B	;	LDIMMLO,NEXT(Stb16_8)
0xe0	ldcode.8 A,(B)	;	TO_Z(R_B),LMAR(1),NEXT(Ldcode8)
0xe1	nop2	;	NEXT(Fetch)
0xe2	stcode.8 (B),A	;	TO_Z(R_B),LMAR(1),NEXT(Stcode8)
0xe3	nop3	;	NEXT(Fetch)
0xe4	enter #fsize16_8	;	LDIMMHI,NEXT(Enter)
0xe5	enter #fsize8	;	NEG1_TO_Z,LATCH(R_MDR),NEXT(Enter)
0xe6	vshl.16 A	;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(Vshl)
0xe7	vshl.16 B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Vshl)
0xe8	memcopy	;	COMPARE_0(R_C),MISC(M_SET_FLAGS),NEXT(Bcopy)
0xe9	tosys	;	PRIV(1),COMPARE_0(R_C),MISC(M_SET_FLAGS),NEXT(ToSys)
0xea	fromsys	;	PRIV(1),COMPARE_0(R_C),MISC(M_SET_FLAGS),NEXT(FromSys)
0xeb	ldclr.8 A,(B)	;	TO_Z(R_B),LMAR(1),NEXT(LdClr)
0xec	wdpte A,(B)	;	PRIV(1),TO_Z(R_B),LMAR(1),NEXT(Wdpte)
0xed	sbc.16 A,B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Sbc16)

0xee	vshr.16 A	;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(Vshr)
0xef	vshr.16 B	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Vshr)
0xf0	cmpb.ne.8 A,#u16(DP),#d8	;	LDIMMHI,NEXT(Cmpb8_indir16)
0xf1	cmpb.ne.8 A,#u8(SP),#d8	;	LDIMMLO,NEXT(Cmpb8_indir)
0xf2	copy A,C	;	TO_Z(R_C),L(R_A,LWORD),NEXT(Fetch)
0xf3	cmpb.ne.8 A,#u8(B),#d8	;	LDIMMLO,NEXT(Cmpb8_indir)
0xf4	cmpb.ne.8 A,#i8_0,#d8	;	LDIMMLO,NEXT(Cmpb8)
0xf5	cmpb.ne.8 A,#0,#d8	;	TO_Z(R_MDR),L(R_MDR,LWORD),NEXT(Cmpb8)
0xf6	copy A,SP	;	TO_Z(R_SP),L(R_A,LWORD),NEXT(Fetch)
0xf7	cmpb.ne.8 A,B,#d8	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Cmpb8)
0xf8	cmpb.ne.16 A,#u16(DP),#d8	;	LDIMMHI,NEXT(Cmpb16_indir16)
0xf9	cmpb.ne.16 A,#u8(SP),#d8	;	LDIMMLO,NEXT(Cmpb16_indir)
0xfa	bkpt	;	MISC(M_BKPT),NEXT(Unreachable)
0xfb	cmpb.ne.16 A,#u8(B),#d8	;	LDIMMLO,NEXT(Cmpb16_indir)
0xfc	cmpb.ne.16 A,#i16_exti8_0,#d8	;	LDIMMHI,NEXT(Cmpb16_16)
0xfd	cmpb.ne.16 A,#exti8_0,#d8	;	LDIMMEXT,NEXT(Cmpb16)
0xfe	cmpb.ne.16 A,#0,#d8	;	TO_Z(R_MDR),L(R_MDR,LWORD),NEXT(Cmpb16)
0xff	cmpb.ne.16 A,B,#d8	;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(Cmpb16)

Top half of PROM - continuation microcode.

0x100	Fetch	;	FETCH_OP
0x101	IRQ5	;	TO_Z(R_MAR),MISC(M_COMMIT),NEXT(Fault)
0x102	IRQ4	;	TO_Z(R_MAR),MISC(M_COMMIT),NEXT(Fault)
0x103	IRQ3	;	TO_Z(R_MAR),MISC(M_COMMIT),NEXT(Fault)
0x104	IRQ2	;	TO_Z(R_MAR),MISC(M_COMMIT),NEXT(Fault)
0x105	IRQ1	;	TO_Z(R_MAR),MISC(M_COMMIT),NEXT(Fault)

0x106	IRQ0	;	TO_Z(R_MAR),MISC(M_COMMIT),NEXT(Fault)
0x107	DMA_req	;	MISC(M_DMA_ACK),NEXT(Fetch)
0x108	Fault_syscall	;	TO_Z(R_MAR),MISC(M_COMMIT),NEXT(Fault)
0x109		;	
0x10a	Fault_ovflo	;	TO_Z(R_MAR),L(R_PC,LWORD),NEXT(Fault)
0x10b	Fault_priv	;	TO_Z(R_MAR),L(R_PC,LWORD),NEXT(Fault)
0x10c	Fault_bkpt	;	TO_Z(R_MAR),L(R_PC,LWORD),NEXT(Fault)
0x10d	Fault_nw	;	TO_Z(R_MAR),L(R_PC,LWORD),NEXT(Fault)
0x10e	Fault_np	;	TO_Z(R_MAR),L(R_PC,LWORD),NEXT(Fault)
0x10f		;	
0x110	Aluop8_indir	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x111		;	LDLO,NEXT(FALLTHRU)
0x112	Aluop8	;	E_L(R_A),E_R(ER_MDR),ALU(OP_IR13,BYTE,NO_CARRY),L(R_A,LBYTE),MISC(M_SET_FLAGS),NEXT(Fetch)
0x113	Aluop8_indir16	;	LDIMMLO,NEXT(Aluop8_indir)
0x114	Aluop16_indir	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x115		;	LDHI,NEXT(FALLTHRU)
0x116		;	LDLO,NEXT(FALLTHRU)
0x117	Aluop16	;	E_L(R_A),E_R(ER_MDR),ALU(OP_IR13,WORD,NO_CARRY),L(R_A,LWORD),MISC(M_SET_FLAGS),NEXT(Fetch)
0x118	Aluop16_indir16	;	LDIMMLO,NEXT(Aluop16_indir)
0x119	Cmp8_indir	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x11a		;	LDLO,NEXT(FALLTHRU)
0x11b	Cmp8	;	E_L(R_A),E_R(ER_MDR),ALU(OP_SUB,BYTE,NO_CARRY),MISC(M_SET_FLAGS),NEXT(Fetch)
0x11c	Cmp8_indir16	;	LDIMMLO,NEXT(Cmp8_indir)
0x11d	Cmp16_indir	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x11e		;	LDHI,NEXT(FALLTHRU)
0x11f		;	LDLO,NEXT(FALLTHRU)
0x120	Cmp16	;	E_L(R_A),E_R(ER_MDR),ALU(OP_SUB,WORD,NO_CARRY),MISC(M_SET_FLAGS),NEXT(Fetch)
0x121	Cmp16_indir16	;	LDIMMLO,NEXT(Cmp16_indir)
0x122	Cmpb8_indir	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x123		;	LDLO,NEXT(FALLTHRU)

0x124	Cmpb8	;	E_L(R_A),E_R(ER_MDR),ALU(OP_SUB,BYTE,NO_CARRY),MISC(M_SET_FLAGS),NEXT(CheckBr)
0x125	Cmpb8_indir16	;	LDIMMLO,NEXT(Cmpb8_indir)
0x126	Cmpb16_indir	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x127		;	LDHI,NEXT(FALLTHRU)
0x128		;	LDLO,NEXT(FALLTHRU)
0x129	Cmpb16	;	E_L(R_A),E_R(ER_MDR),ALU(OP_SUB,WORD,NO_CARRY),MISC(M_SET_FLAGS),NEXT(CheckBr)
0x12a	Cmpb16_indir16	;	LDIMMLO,NEXT(Cmpb16_indir)
0x12b	CheckBr	;	LDIMMEXT,CBR(B_NORMAL,TakenBr)
0x12c	TakenBr	;	E_L(R_PC),E_R(ER_MDR),ALU(OP_ADD,WORD,NO_CARRY),L(R_PC,LWORD),LMAR(1),NEXT(Fetch)
0x12d	BrNormal	;	LDIMMLO,CBR(B_NORMAL,TakenBr)
0x12e	BrNegated	;	LDIMMLO,CBR(B_NEGATED,TakenBr)
0x12f	Bset8	;	E_L(R_A),E_R(ER_MDR),ALU(OP_AND,BYTE,NO_CARRY),MISC(M_SET_FLAGS),NEXT(CheckBrNeg)
0x130	CheckBrNeg	;	LDIMMEXT,CBR(B_NEGATED,TakenBr)
0x131	Bclr8	;	E_L(R_A),E_R(ER_MDR),ALU(OP_AND,BYTE,NO_CARRY),MISC(M_SET_FLAGS),NEXT(CheckBr)
0x132	Bset16	;	LDIMMLO,NEXT(FALLTHRU)
0x133		;	E_L(R_A),E_R(ER_MDR),ALU(OP_AND,WORD,NO_CARRY),MISC(M_SET_FLAGS),NEXT(CheckBrNeg)
0x134	Bclr16	;	LDIMMLO,NEXT(FALLTHRU)
0x135		;	E_L(R_A),E_R(ER_MDR),ALU(OP_AND,WORD,NO_CARRY),MISC(M_SET_FLAGS),NEXT(CheckBr)
0x136	Push16	;	DEC_TO_Z(R_SP),LMAR(1),NEXT(FALLTHRU)
0x137		;	WRITELO,DEC_TO_Z(R_MAR),LMAR(1),L(R_SP,LWORD),NEXT(FALLTHRU)
0x138		;	WRITEHI,TO_Z(R_PC),LMAR(1),NEXT(Fetch)
0x139	Pop16	;	LDHI,NEXT(FALLTHRU)
0x13a		;	READLO,INC_TO_Z(R_MAR),L(R_SP,LWORD),LMAR(1),NEXT(FALLTHRU)
0x13b		;	TO_Z(R_MDR),L(R_IR_REG,LWORD),NEXT(FALLTHRU)
0x13c		;	TO_Z(R_PC),LMAR(1),NEXT(Fetch)
0x13d	Lda8_8	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x13e		;	LDLO,NEXT(FALLTHRU)
0x13f		;	TO_Z(R_MDR),L(R_A,LBYTE),NEXT(Fetch)

0x140	Lda8_16	;	LDIMMLO,NEXT(Lda8_8)
0x141	Ldb8_8	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x142		;	LDLO,NEXT(FALLTHRU)
0x143		;	TO_Z(R_MDR),L(R_B,LBYTE),NEXT(Fetch)
0x144	Ldb8_16	;	LDIMMLO,NEXT(Ldb8_8)
0x145	Lda16_8	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x146		;	LDHI,NEXT(FALLTHRU)
0x147		;	LDLO,NEXT(FALLTHRU)
0x148		;	TO_Z(R_MDR),L(R_A,LWORD),NEXT(Fetch)
0x149	Lda16_16	;	LDIMMLO,NEXT(Lda16_8)
0x14a	Ldb16_8	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x14b		;	LDHI,NEXT(FALLTHRU)
0x14c		;	LDLO,NEXT(FALLTHRU)
0x14d		;	TO_Z(R_MDR),L(R_B,LWORD),NEXT(Fetch)
0x14e	Ldb16_16	;	LDIMMLO,NEXT(Ldb16_8)
0x14f	Sta8_8	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x150		;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(FALLTHRU)
0x151	StaLo	;	STLO,NEXT(Fetch)
0x152	Sta8_16	;	LDIMMLO,NEXT(Sta8_8)
0x153	Sta16_8	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x154		;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(FALLTHRU)
0x155		;	STHI,NEXT(StaLo)
0x156	Sta16_16	;	LDIMMLO,NEXT(Sta16_8)
0x157	Stb8_8	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x158		;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(FALLTHRU)
0x159	StbLo	;	STLO,NEXT(Fetch)
0x15a	Stb8_16	;	LDIMMLO,NEXT(Stb8_8)
0x15b	Stb16_8	;	GEN_ADDR(R_IR_BASE),LMAR(1),NEXT(FALLTHRU)
0x15c		;	TO_Z(R_B),L(R_MDR,LWORD),NEXT(FALLTHRU)
0x15d		;	STHI,NEXT(StbLo)
0x15e	Stb16_16	;	LDIMMLO,NEXT(Stb16_8)
0x15f	Sbc16	;	E_L(R_A),E_R(ER_MDR),ALU(OP_SUB,WORD,CARRY_IN),L(R_A,LWORD),MISC(M_SET_FLAGS),NEXT(Fetch)

0x160	Adc16	;	E_L(R_A),E_R(ER_MDR),ALU(OP_ADD,WORD,CARRY_IN),L(R_A,LWORD),MISC(M_SET_FLAGS),NEXT(Fetch)
0x161	LdaA_16	;	LDIMMLO,NEXT(LdaA)
0x162	LdaA	;	GEN_ADDR(R_IR_BASE),L(R_A,LWORD),NEXT(Fetch)
0x163	LdaB_16	;	LDIMMLO,NEXT(LdaB)
0x164	LdaB	;	GEN_ADDR(R_IR_BASE),L(R_B,LWORD),NEXT(Fetch)
0x165	LdiA8	;	TO_Z(R_MDR),L(R_A,LBYTE),NEXT(Fetch)
0x166	LdiB8	;	TO_Z(R_MDR),L(R_B,LBYTE),NEXT(Fetch)
0x167	LdiA16_lo	;	LDIMMLO,NEXT(LdiA16)
0x168	LdiA16	;	TO_Z(R_MDR),L(R_A,LWORD),NEXT(Fetch)
0x169	LdiB16_lo	;	LDIMMLO,NEXT(LdiB16)
0x16a	LdiB16	;	TO_Z(R_MDR),L(R_B,LWORD),NEXT(Fetch)
0x16b	LdiC16_lo	;	LDIMMLO,NEXT(LdiC16)
0x16c	LdiC16	;	TO_Z(R_MDR),L(R_C,LWORD),NEXT(Fetch)
0x16d	RelBrLo	;	LDIMMLO,NEXT(RelBr)
0x16e	RelBr	;	GEN_ADDR(R_PC),L(R_PC,LWORD),LMAR(1),NEXT(Fetch)
0x16f	CallImm	;	DEC_TO_Z(R_SP),LMAR(1),NEXT(FALLTHRU)
0x170		;	WRITELO,DEC_TO_Z(R_MAR),LMAR(1),L(R_SP,LWORD),NEXT(FALLTHRU)
0x171		;	WRITEHI,TO_Z(R_PC),LMAR(1),NEXT(FALLTHRU)
0x172		;	LDIMMHI,NEXT(FALLTHRU)
0x173		;	LDIMMLO,NEXT(FALLTHRU)
0x174		;	GEN_ADDR(R_PC),L(R_PC,LWORD),LMAR(1),NEXT(Fetch)
0x175	CallA	;	DEC_TO_Z(R_SP),LMAR(1),NEXT(FALLTHRU)
0x176		;	WRITELO,DEC_TO_Z(R_MAR),LMAR(1),L(R_SP,LWORD),NEXT(FALLTHRU)
0x177		;	WRITEHI,TO_Z(R_A),L(R_PC,LWORD),LMAR(1),NEXT(Fetch)
0x178	LdClr	;	READLO,NEXT(FALLTHRU)
0x179		;	TO_Z(R_MDR),L(R_A,LBYTE),NEXT(FALLTHRU)
0x17a		;	WRITEHI,TO_Z(R_PC),LMAR(1),NEXT(Fetch)
0x17b	Wcpte	;	USER_PTB(1),CODE_PTB(1),E_L(R_A),MISC(M_LPTE),NEXT(PCtoMAR)
0x17c	Enter	;	LDIMMLO,NEXT(FALLTHRU)
0x17d		;	GEN_ADDR(R_SP),LMAR(1),NEXT(FALLTHRU)
0x17e		;	TO_Z(R_SP),L(R_MDR,LWORD),NEXT(FALLTHRU)

0x17f		;	DEC_TO_Z(R_MAR),LMAR(1),NEXT(FALLTHRU)
0x180		;	WRITELO,DEC_TO_Z(R_MAR),LMAR(1),L(R_SP,LWORD),NEXT(FALLTHRU)
0x181		;	WRITEHI,TO_Z(R_MAR),L(R_SP,LWORD),NEXT(FALLTHRU)
0x182		;	TO_Z(R_PC),LMAR(1),NEXT(Fetch)
0x183	Bcopy	;	CBR(B_NEGATED,FALLTHRU)
0x184		;	TO_Z(R_B),LMAR(1),NEXT(FALLTHRU)
0x185		;	READLO,TO_Z(R_A),LMAR(1),NEXT(Bcopy0)
0x186	ToSys	;	PRIV(1),CBR(B_NEGATED,FALLTHRU)
0x187		;	TO_Z(R_B),LMAR(1),NEXT(FALLTHRU)
0x188		;	USER_PTB(1),READLO,TO_Z(R_A),LMAR(1),NEXT(FALLTHRU)
0x189	Bcopy0	;	WRITELO,INC_TO_Z(R_MAR),L(R_A,LWORD),NEXT(FALLTHRU)
0x18a	Bcopy1	;	INC_TO_Z(R_B),L(R_B,LWORD),NEXT(FALLTHRU)
0x18b		;	DEC_TO_Z(R_C),L(R_C,LWORD),NEXT(FALLTHRU)
0x18c	BackupPC	;	DEC_TO_Z(R_PC),L(R_PC,LWORD),LMAR(1),NEXT(Fetch)
0x18d	FromSys	;	PRIV(1),CBR(B_NEGATED,FALLTHRU)
0x18e		;	TO_Z(R_B),LMAR(1),NEXT(FALLTHRU)
0x18f		;	READLO,TO_Z(R_A),LMAR(1),NEXT(FALLTHRU)
0x190		;	USER_PTB(1),WRITELO,INC_TO_Z(R_MAR),L(R_A,LWORD),NEXT(Bcopy1)
0x191	Fault	;	DEC_TO_Z(R_SSP),LMAR(1),NEXT(FALLTHRU)
0x192		;	TO_MDR(R_MSW),NEXT(FALLTHRU)
0x193		;	ZERO_TO_Z,MISC(M_LEI),LMODE(1),NEXT(FALLTHRU)
0x194		;	PUSHLO,NEXT(FALLTHRU)
0x195		;	PUSHHI,NEXT(FALLTHRU)
0x196		;	TO_MDR(R_SP),NEXT(FALLTHRU)
0x197		;	PUSHLO,NEXT(FALLTHRU)
0x198		;	PUSHHI,NEXT(FALLTHRU)
0x199		;	TO_MDR(R_TPC),NEXT(FALLTHRU)
0x19a		;	PUSHLO,NEXT(FALLTHRU)
0x19b		;	PUSHHI,NEXT(FALLTHRU)
0x19c		;	TO_MDR(R_A),NEXT(FALLTHRU)
0x19d		;	PUSHLO,NEXT(FALLTHRU)
0x19e		;	PUSHHI,NEXT(FALLTHRU)

0x19f		;	TO_MDR(R_B),NEXT(FALLTHRU)
0x1a0		;	PUSHLO,NEXT(FALLTHRU)
0x1a1		;	PUSHHI,NEXT(FALLTHRU)
0x1a2		;	TO_MDR(R_C),NEXT(FALLTHRU)
0x1a3		;	PUSHLO,NEXT(FALLTHRU)
0x1a4		;	PUSHHI,NEXT(FALLTHRU)
0x1a5		;	TO_MDR(R_DP),NEXT(FALLTHRU)
0x1a6		;	PUSHLO,L(R_SP,LWORD),NEXT(FALLTHRU)
0x1a7		;	PUSHHI,NEXT(FALLTHRU)
0x1a8		;	TO_Z(R_PC),L(R_A,LWORD),NEXT(FALLTHRU)
0x1a9		;	TO_Z(R_FCODE),LMAR(1),NEXT(FALLTHRU)
0x1aa		;	POPHI,NEXT(FALLTHRU)
0x1ab		;	POPLO,NEXT(FALLTHRU)
0x1ac		;	FROM_MDR(R_PC),LMAR(1),MISC(M_CLR_TRAP),NEXT(Fetch)
0x1ad	Reti	;	TO_Z(R_SP),LMAR(1),NEXT(FALLTHRU)
0x1ae		;	POPHI,NEXT(FALLTHRU)
0x1af		;	POPLO,NEXT(FALLTHRU)
0x1b0		;	FROM_MDR(R_DP),NEXT(FALLTHRU)
0x1b1		;	POPHI,NEXT(FALLTHRU)
0x1b2		;	POPLO,NEXT(FALLTHRU)
0x1b3		;	FROM_MDR(R_C),NEXT(FALLTHRU)
0x1b4		;	POPHI,NEXT(FALLTHRU)
0x1b5		;	POPLO,NEXT(FALLTHRU)
0x1b6		;	FROM_MDR(R_B),NEXT(FALLTHRU)
0x1b7		;	POPHI,NEXT(FALLTHRU)
0x1b8		;	POPLO,NEXT(FALLTHRU)
0x1b9		;	FROM_MDR(R_A),NEXT(FALLTHRU)
0x1ba		;	POPHI,NEXT(FALLTHRU)
0x1bb		;	POPLO,NEXT(FALLTHRU)
0x1bc		;	FROM_MDR(R_PC),NEXT(FALLTHRU)
0x1bd		;	POPHI,NEXT(FALLTHRU)
0x1be		;	POPLO,NEXT(FALLTHRU)
0x1bf		;	TO_Z(R_MDR),MISC(M_COMMIT),NEXT(FALLTHRU)

0x1c0		;	POPHI,NEXT(FALLTHRU)
0x1c1		;	READLO,INC_TO_Z(R_MAR),L(R_SP,LWORD),NEXT(FALLTHRU)
0x1c2		;	TO_Z(R_MDR),L(R_MSW,LWORD),LMODE(1),LPAGING(1),MISC(M_LEI),NEXT(FALLTHRU)
0x1c3		;	TO_Z(R_TPC),L(R_SP,LWORD),NEXT(FALLTHRU)
0x1c4		;	TO_Z(R_PC),LMAR(1),NEXT(Fetch)
0x1c5	Syscall	;	TO_Z(R_MDR),L(R_A,LWORD),NEXT(FALLTHRU)
0x1c6		;	MISC(M_SYSCALL),NEXT(Unreachable)
0x1c7	Ldcode8	;	CODE_PTB(1),LDLO,NEXT(FALLTHRU)
0x1c8		;	TO_Z(R_MDR),L(R_A,LBYTE),NEXT(Fetch)
0x1c9	Stcode8	;	TO_Z(R_A),L(R_MDR,LWORD),NEXT(Stcodelo)
0x1ca	Stcodelo	;	CODE_PTB(1),STLO,NEXT(Fetch)
0x1cb	Wdpte	;	USER_PTB(1),E_L(R_A),MISC(M_LPTE),NEXT(PCtoMAR)
0x1cc	Shlb16	;	E_L(R_B),E_R(ER_MDR),ALU(OP_ADD,WORD,NO_CARRY),L(R_B,LWORD),NEXT(Fetch)
0x1cd	Shla16	;	E_L(R_A),E_R(ER_MDR),ALU(OP_ADD,WORD,NO_CARRY),L(R_A,LWORD),NEXT(Fetch)
0x1ce	Aluop16_16	;	LDIMMLO,NEXT(Aluop16)
0x1cf	Cmpb16_16	;	LDIMMLO,NEXT(Cmpb16)
0x1d0	Cmp16_16	;	LDIMMLO,NEXT(Cmp16)
0x1d1	PCtoMAR	;	TO_Z(R_PC),LMAR(1),NEXT(Fetch)
0x1d2	Vshl	;	COMPARE_0(R_C),MISC(M_SET_FLAGS),NEXT(FALLTHRU)
0x1d3		;	CBR(B_NEGATED,FALLTHRU)
0x1d4		;	COMPARE_0(R_MDR),MISC(M_SET_FLAGS),NEXT(FALLTHRU)
0x1d5		;	CBR(B_NEGATED,FALLTHRU)
0x1d6		;	E_L(R_MDR),E_R(ER_MDR),ALU(OP_ADD,WORD,NO_CARRY),L(R_IR_REG,LWORD),NEXT(Bcopy1)
0x1d7	Vshr	;	COMPARE_0(R_C),MISC(M_SET_FLAGS),NEXT(FALLTHRU)
0x1d8		;	CBR(B_NEGATED,FALLTHRU)
0x1d9		;	COMPARE_0(R_MDR),MISC(M_SET_FLAGS),NEXT(FALLTHRU)
0x1da		;	CBR(B_NEGATED,FALLTHRU)
0x1db		;	TO_Z(R_MDR),MISC(M_RSHIFT),L(R_IR_REG,LWORD),NEXT(Bcopy1)
0x1dc	LeaPC	;	LDIMMLO,NEXT(FALLTHRU)
0x1dd		;	GEN_ADDR(R_PC),L(R_IR_REG,LWORD),NEXT(Fetch)
0x1de	LeaAAB2	;	E_R(ER_MDR),E_L(R_MDR),ALU(OP_ADD,WORD,NO_CARRY),L(R_MDR,L

			WORD),NEXT(FALLTHRU)
0x1df	LeaA1	;	E_R(ER_MDR),E_L(R_A),ALU(OP_ADD,WORD,NO_CARRY),L(R_A,LWORD),NEXT(Fetch)
0x1e0	LeaBBA2	;	E_R(ER_MDR),E_L(R_MDR),ALU(OP_ADD,WORD,NO_CARRY),L(R_MDR,LWORD),NEXT(FALLTHRU)
0x1e1	LeaB1	;	E_R(ER_MDR),E_L(R_B),ALU(OP_ADD,WORD,NO_CARRY),L(R_B,LWORD),NEXT(Fetch)
0x1e2	LeaABA2	;	E_R(ER_MDR),E_L(R_MDR),ALU(OP_ADD,WORD,NO_CARRY),L(R_MDR,LWORD),NEXT(FALLTHRU)
0x1e3		;	E_R(ER_MDR),E_L(R_B),ALU(OP_ADD,WORD,NO_CARRY),L(R_A,LWORD),NEXT(Fetch)
0x1e4	LeaBAB2	;	E_R(ER_MDR),E_L(R_MDR),ALU(OP_ADD,WORD,NO_CARRY),L(R_MDR,LWORD),NEXT(FALLTHRU)
0x1e5		;	E_R(ER_MDR),E_L(R_A),ALU(OP_ADD,WORD,NO_CARRY),L(R_B,LWORD),NEXT(Fetch)
0x1e6	CopyMSWA	;	TO_Z(R_A),L(R_MSW,LWORD),LMODE(1),LPAGING(1),MISC(M_LEI),NEXT(FALLTHRU)
0x1e7		;	NEXT(Fetch)
0x1e8	Strcopy	;	READLO,TO_Z(R_A),LMAR(1),NEXT(FALLTHRU)
0x1e9		;	WRITELO,COMPARE8_0(R_MDR),MISC(M_SET_FLAGS),NEXT(FALLTHRU)
0x1ea		;	TO_Z(R_PC),LMAR(1),NEXT(FALLTHRU)
0x1eb		;	INC_TO_Z(R_A),L(R_A,LWORD),NEXT(FALLTHRU)
0x1ec		;	INC_TO_Z(R_B),L(R_B,LWORD),CBR(B_NEGATED,BackupPC)
0x1ed			
0x1ee			
0x1ef			
0x1f0			
0x1f1			
0x1f2		;	
0x1f3		;	
0x1f4		;	
0x1f5		;	
0x1f6		;	
0x1f7		;	
0x1f8		;	
0x1f9		;	
0x1fa		;	

0x1fb		;	
0x1fc		;	
0x1fd		;	
0x1fe	Unreachable	;	NEXT(Unreachable)
0x1ff	**UNUSABLE**	;	

```
// ENDPREPROCESS prombits.h
```