An Approach to Microcomputer Architecture with AMD Bit-Slice Components

Joe Parks Allen Trentham

December 5, 1991

Introduction

The primary objective of this project was to expose the student to more intricate digital system design with an emphasis in computer microarchitecture. A microprocessor, with peripherals, was designed to meet all specifications while being able to handle any forthcoming expansion. An extensive instruction set was also developed in order to make the unit useful for a variety of purposes. The original specifications outlined several components. All of these were implemented with a few additions, as indicated.

Communications Bus:

A communications bus with 16-bit data and address buses was constructed. The driving capabilities of the buses were designed so additional components could be added without overloading.

Microprocessor Module:

A 16-bit single address CPU including fetch, execute, reset, and interrupt routines was created. This device allowed for immediate, direct, indirect, extended, indexed, and inherent addressing modes. Means for employing an external stack were also included. Six external interrupt request lines are available for the user.

Memory Modules:

RAM and EPROM modules were designed to allow plenty of room for a variety of different applications. Each module contains 8K words and is addressable at the low and high end of memory, respectively.

I/O Modules:

A variety of memory referenced I/O modules was included in the CPU. Decoding for up to 22 additional units has already been implemented for expendability. The modules include:

1. A 16-bit LED driver for output
2. A 16-bit switch input
3. A 16-bit solid state relay driver
4. A 16-bit general purpose bit-selectable I/O port
5. A dual 12-bit DAC
6. A 12-bit ADC
7. An asynchronous serial communications interface

This report is divided into two parts, hardware and software. The software portion includes all macro and micro instructions while the hardware section deals with the physical devices.

The CPU : A Hardware Description

The central processing unit was designed using AMD bit-slice processor chips. The devices were designed to be used in a way similar to the requirements of this project therefore much of the layout strategy was derived from examples in the data books.

The CPU can essentially be divided into two parts, the computer control unit and the arithmetic/logic unit. The computer control unit's (CCU) primary task is to carry out program execution. All macroinstructions are decoded and executed within this unit. This execution employs an internal memory which contains the control sequences for the entire instruction set. The arithmetic/logic unit (ALU) performs all operations on memory and is completely dependant on the CCU for control. The memory address (MAR) and the memory buffer (MBR) registers are also located within the ALU. These registers are used to transfer data to and from external memory. Both units are connected to the main data bus while only the ALU is connected to the main address bus. This design mandates that all main memory address pointers lie within internal registers in the ALU.

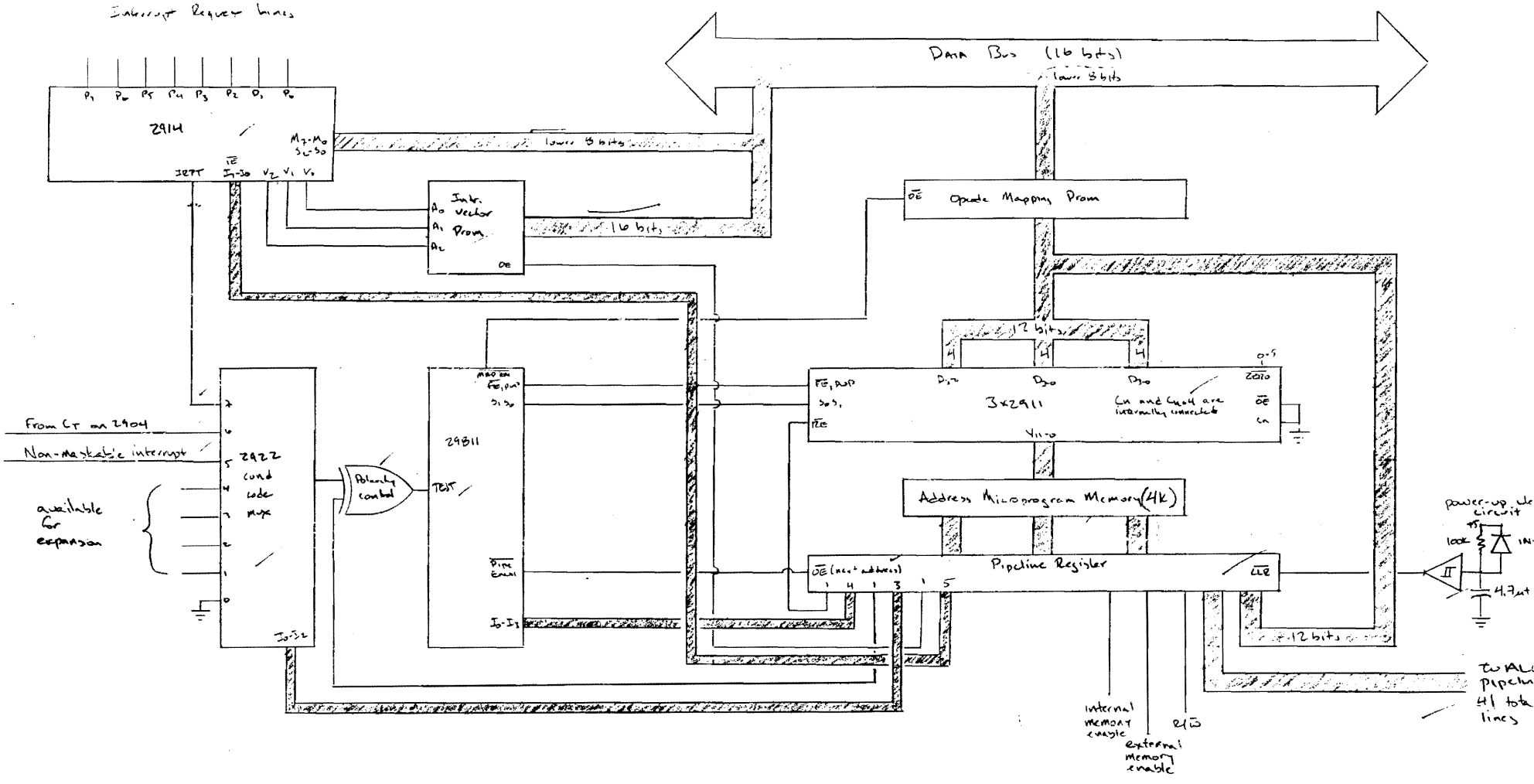
The CCU:

The CCU was designed around six basic components; the microprogram sequencer, the next address control unit, the priority interrupt decoder, the opcode mapping PROM, the microprogram memory, and the pipeline register. Each of these parts plays a very active role in the operation of the microprocessor. Reference

Diagram 1 for the following discussion.

The opcode mapping PROM reads a value from the data bus which corresponds to a macroinstruction. The opcodes will be defined in greater detail later in the report. The mapping PROM uses this opcode as an address from which it produces, if selected, a twelve bit word which is entered into the microprogram sequencer. This word is simply the starting address of the microprogram which performs that specific macroinstruction. The output enable of this PROM is regulated by the next address control unit.

The twelve bit address, which could have originated in the opcode mapping PROM, is channeled through the microprogram sequencer. This sequencer is composed of three Am2911 devices cascaded together. The 2911 was designed specifically for allowing multiple way branching within the microprogram. The sequencer can select between four possible addresses. These addresses include direct inputs from an external source, a four word deep stack, an internal address register which may contain a previous address entered from the direct inputs, and an internal program counter which contains the last address incremented by one. The 2911 selects one of these four addresses through a 4-1 multiplexer which is controlled externally by S_0 and S_1. Operations on the internal stack are performed by the FE* and PUP control lines. These operations include pushing and popping item to the stack. All four of these control lines are connected to the next address control unit. Table 1 illustrates the effects of these control lines. An



Computer Control Unit with 4K program memory
Priority interrupt decoder included
Diagram 1

additional control line , RE*, enables the internal address register to latch the value present on the direct address lines. This latch occurs at the LOW to HIGH transition of the system clock. The RE* control line is connected to the pipeline register.

Table 1.

S_1	S_0	OUTPUT
L	L	microprogram counter (uPC)
L	H	address/ holding register (AR)
H	L	read from stack (F)
H	H	direct inputs (D)

FE*	PUP	STACK OPERATION
L	L	pop stack (decrement pointer)
L	H	increment stack pointer and push current address onto stack
H	X	no change

The address which is present at the output of the 2911 is fed directly into the microprogram memory. The microprogram memory is 4K ROM memory with an output word which is 73 bits wide. All of the control sequences for each of the microprocessor's operations are stored here. This memory is always engaged and its output is connected directly to the pipeline register.

The pipeline register is nothing more than a series of D - type flip flops. When clocked (low to high transition), the output of the microprogram memory is latched and becomes the output of the

pipeline. Twelve bits of the pipeline are designated as the branch address. These lines connect back to the direct inputs of the 2911. To avoid any bus contention with the opcode mapping PROM, a three state buffer with an external output enable is connected to these twelve bits. The pipeline output enable signal originates from the next address control unit. The pipeline also has an external clear line. As a power - up consideration, this clear line is connected to a non-inverting Schmitt trigger with a simple R-C network as its input which forces the pipeline register low during reset. This has the effect of initializing the machine to a known state which starts the microprogram at the desired location (address zero). The outputs of the pipeline directly correspond to all of the control lines used by the entire microprocessor. Table 2 introduces the pipeline register. The control lines will all be covered in further detail later in the report.

Table 2

BITS	CONTROL NAME	DESCRIPTION
0 - 11	A_0-A_{11}	branch address
12 - 15	I_0-I_3	instruction set for Am29811
16 - 18	I_0-I_2	condition code mux select
19	pol	polarity control (0-pass, 1-invert)
20 -23	I_0-I_3	instruction set for Am2914
24	I_E*	Am2914 instruction enable (low when reading instruction)
25	E_N	interrupt vector PROM enable
26	RE*	register enable for Am2911

Table 2 cont.

BITS	CONTROL NAME	DESCRIPTION
27	IME*	internal memory enable (only used during power-up)
28	EME	external memory enable (for normal memory addressing)
29	R/W*	read/write control line for memory
30	MAR	memory address register enable
31	RLE*	MBR receiver latch enable
32	RE*	MBR bus enable
33 - 44	I_0-I_9, I_{11}-I_{12}	instruction set for Am2904 (I_{10} is connected to I_8 on Am2903)
45	OE_Y*	MSR output register enable on Am2904
46	CE_M*	MSR input enable on Am2904
47	E_{OVR}*	MSR overflow bit enable on Am2904
48	E_C*	MSR carry bit enable on Am2904
49	E_N*	MSR sign bit enable on Am2904
50	E_Z*	MSR zero bit enable on Am2904
51	CE_u*	uSR input enable on Am2904
52 - 61	I_0-I_4, I_{5a}, I_{5b}, I_6-I_8	instruction set for Am2903 (I_{5a} connects to lower slices I_{5b} connects to upper slices)
62 - 65	B_0-B_3	RAM B address lines for Am2903
66 - 69	A_0-A_3	RAM A address lines for Am2903
70	OE_{Y2}*	output enable for port Y_{0-3} on Am2903
71	OE_B*	RAM or DB port select on Am2903
72	E_A*	RAM or DA port select on Am2903

The next address controller (Am29811) is an essential part of the microprocessor. This device regulates all of the control lines to the 2911s and to the pipeline and mapping PROM. Control of the 29811 comes from two sources, the pipeline and a condition test bit. Four control lines (I_0 - I_3) originating from the pipeline provide sixteen different instructions. Some of these instructions are never used because an internal event counter wasn't implemented in this design and many of these instructions are conditional meaning that the test bit is monitored. Table 3 illustrates the available instructions for the 29811. An 8-1 multiplexer, in addition to a polarity control gate, regulates which source will become the test input. The possible test lines include an IRQ request from the priority interrupt controller, a condition set by the status control unit (in the ALU) , a non-maskable interrupt, and a logic low or high. There are still four additional test lines available for future expansion. Table 4 shows the control of the test input line.

Table 3

FUNCTION	INPUTS					NEXT ADDR		FILE		MAPE*	PIPE*
	I_3	I_2	I_1	I_0	TEST	S_1	S_0	FE*	PUP		
jump zero	L	L	L	L	L	H	H	H	H	H	L
	L	L	L	L	H	H	H	H	H	H	L
conditional	L	L	L	H	L	L	L	H	H	H	L
jump to sub-routine	L	L	L	H	H	H	H	L	H	H	L
jump to mapping	L	L	H	L	L	H	H	H	H	L	H
PROM output	L	L	H	L	H	H	H	H	H	L	H

Table 3 cont.

FUNCTION	INPUTS I_3	I_2	I_1	I_0	TEST	NEXT ADDR S_1	S_0	FILE FE*	PUP	MAPE*	PIPE*
cond jump addr.	L	L	H	H	L	L	L	H	H	H	L
in pipeline	L	L	H	H	H	H	H	H	H	H	L
jump to sub-	L	H	L	H	L	L	H	L	H	H	L
routine addr.	L	H	L	H	H	H	H	L	H	H	L
in AR or addr in pipeline											
cond jump to	L	H	H	L	L	L	L	H	H	H	L
vector in PL	L	H	H	L	H	H	H	H	H	H	L
jump to addr	L	H	H	H	L	L	H	H	H	H	L
in AR or addr	L	H	H	H	H	H	H	H	H	H	L
in pipeline											
cond. return	H	L	H	L	L	L	L	H	L	H	L
from subroutine	H	L	H	L	H	H	L	L	L	H	L
cond. jump to	H	L	H	H	L	L	L	H	L	H	L
pipeline and	H	L	H	H	H	H	H	L	L	H	L
pop stack											
continue to	H	H	H	L	L	L	L	H	H	H	L
next address	H	H	H	L	H	L	L	H	H	H	L
jump to addr	H	H	H	H	L	H	H	H	H	H	L
in pipeline	H	H	H	H	H	H	H	H	H	H	L

Table 4

I_2	I_1	I_0	TEST OUTPUT
L	L	L	logic zero
H	H	H	IRQ from priority interrupt decoder
H	H	L	TEST bit from status control unit
H	L	H	non-maskable interrupt
OTHER			open for future expansion

The final component in the CCU is the priority interrupt decoder (Am2914). This device can handle up to eight interrupts, either pulses or levels. Each interrupt line is individually maskable and the chip provides operations for handling the mask or it may be loaded directly from the low order byte of the data bus. An internal status register, which may also be loaded from the data bus, keeps track of the lowest level of interrupt which can drive the IRQ* line. The output of the device is a 3-bit vector which is connected to an interrupt vector ROM. The system interrupts are enabled and disabled by issuing a command to the 2914. Once the interrupts are enabled, all non-masked interrupts will cause the IRQ line, attached to the test bit on the 29811, to be pulled low. This test bit is polled directly before each opcode is fetched to insure prompt interrupt service. If an interrupt is present, the vector output of the 2914 is enabled through control lines on the pipeline. This 3-bit vector is the address of the interrupt vector PROM which produces a 16-bit word onto the data bus. This word is then used by the ALU to fetch the starting location of the interrupt service routine. Once the service routine is completed, the 2914 must be cleared as well as the device which originally caused the interrupt. More details concerning interrupt service, the mask register, and the status register will be discussed in the interrupt service portion of this report. Table 5 lists the possible commands to the 2914.

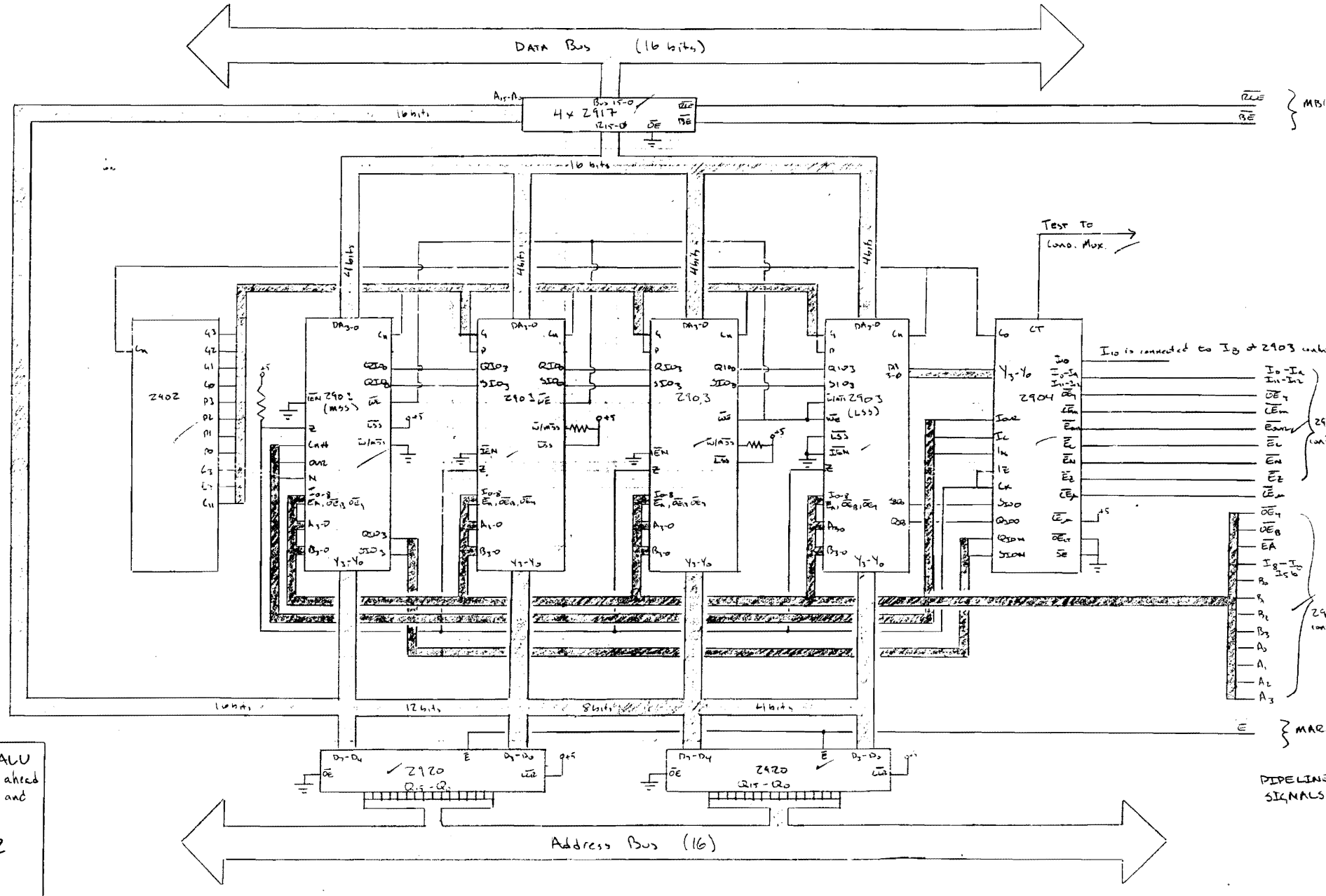
Table 5

I_3	I_2	I_1	I_0	INSTRUCTION
L	L	L	L	clears all interrupts, mask register, status register, and interrupt enable
L	L	L	H	clear all interrupts
L	L	H	L	clear interrupts from M bus data
L	L	H	H	clear interrupts from mask register
L	H	L	L	clear interrupt of last vector read
L	H	L	H	read vector output, inc. status reg.
L	H	H	L	read status register to S bus
L	H	H	H	read mask register to M bus
H	L	L	L	set mask register (all bits)
H	L	L	H	load status register from S bus
H	L	H	L	bit clear mask register from M bus
H	L	H	H	bit set mask register from M bus
H	H	L	L	clear mask register (all bits)
H	H	L	H	disable interrupt request
H	H	H	L	load mask register from M bus
H	H	H	H	enable interrupt request

The ALU:

The ALU and memory control registers are probably the most complex part of the processor. This is true primarily because of the extraordinary control abilities of the status and shift control unit and bit-slice microprocessor device. Reference Diagram 2 during the following discussion.

The heart of the ALU is the actual processor. The Am2903 bit-slice device was selected due to its excellent instruction set and its ability to interface with other devices. Four of these units were cascaded together to form a 16-bit ALU. There are sixteen internal registers within the 2903 which can be accessed by either the B_3 - B_0 or the A_3 - A_0 RAM address inputs. The actual configuration of the internal RAM registers will be discussed in the microprogram section of this report. There are also two external ports from which data may be directly entered from the bus. The DA_{0-3} port may be used as an input instead of the register addressed by A_3 - A_0. In this design the DA_{0-3} port is used to enter information from the data bus. The E_A* control line determines which input is used. When E_A* is high, the DA_{0-3} port is selected. The DB_{0-3} port is a three state input/output port and is connected to the output of the status control unit (lower slice only). Through the use of the OE_B* and I_0 control pins, one of three different inputs can be used as the second input to the ALU. Either the register addressed by B_3 - B_0, the inputs from the DB_{0-3} port, or an internal "Q" register used by the 2903's special functions can be used as the input. Table 6 is a truth table for



16 bit ALU with look ahead carry. MAR and MBR shown

Diagram 2

the operand sources. A write enable (WE*) pin is used to reroute the ALU output back to the register addressed by A_3 - A_0. When this pin is low, the output is written to the RAM address.

Table 6

E_A*	I_0	OE_B*	OPERAND 1	OPERAND 2
L	L	L	RAM @ A_3 - A_0	RAM @ B_3 - B_0
L	L	H	RAM @ A_3 - A_0	DB_{0-3}
L	H	X	RAM @ A_3 - A_0	Q register
H	L	L	DA_{0-3}	RAM @ B_3 - B_0
H	L	H	DA_{0-3}	DB_{0-3}
H	H	X	DA_{0-3}	Q register

Each of the four chips must be configured to be a least significant slice, an intermediate slice, or a most significant slice. This is done by tying the LSS* and WRITE*/MSS* pins high or low according to Table 7. The WRITE*/MSS* pin must be tied high through a current limiting resistor.

Table 7

SLICE ORDER	LSS*	WRITE*/MSS*
least significant	LOW	becomes write line
intermediate	HIGH	HIGH
most significant	HIGH	LOW

The carry generate (G*/N) and the carry propagate (P*/OVR) pins are connected to the look ahead carry generator for all but the most significant slice. For the most significant slice, the sign bit (G*/N) and the overflow indicator (P*/OVR) pins are connected to the status control unit. The zero detect pins (Z) are all wire-ORed together and are also connected to the status control register. The carry-in pin (C_n) pin is connected to the status control unit for the least significant slice and is connected to the look ahead carry unit for the three other slices. The carry-out pin (C_{n+4}) is only used for the most significant slice where it becomes the carry-out detect. The serial shift inputs and outputs for the ALU and the Q register (SIO_0, SIO_3, QIO_0, QIO_3) are cascaded together in between slices and are connected to the shift control unit at the most and least significant slices (see diagram).

Data may be exported from the ALU through the Y_{0-3} port which is connected to the MAR and the MBR registers. The output drivers for this port are engaged when OE_Y* is pulled low.

The control lines I_0-I_4, I_{5a}, I_{5b}, I_6-I_8 determine which operation is to be performed by the 2903. The two I_5 lines allow for sign extension over the upper byte of the ALU. I_{5a} is connected to the lower to slices while I_{5b} is connected to the upper slices. Table 8 introduces the valid ALU operations. All of these will be fully explained when implemented.

Table 8

I_8-I_1	OPERATION
LLLLLLLH	$F = S - R - 1 + C_n$
LLLLLLHL	$F = R - S - 1 + C_n$
LLLLLLHH	$F = R + S + C_n$
LLLLLHLL	$F = S + C_n$
LLLLLHLH	$F = S^* + C_n$
LLLLLHHL	$F = R + C_n$
LLLLLHHH	$F = R^* + C_n$
LLLLHLLL	F_i = LOW
LLLLHLLH	$F_i = R_i^*$ AND S_i
LLLLHLHL	$F_i = R_i$ EXCLUSIVE NOR S_i
LLLLHLHH	$F_i = R_i$ EXCLUSIVE OR S_i
LLLLHHLL	$F_i = R_i$ AND S_i
LLLLHHLH	$F_i = R_i$ NOR S_i
LLLLHHHL	$F_i = R_i$ NAND S_i
LLLLHHHH	$F_i = R_i$ OR S_i
LLLLLLLL	unsigned multiply
LLHLLLLL	two's complement multiply
LHLLLLLL	increment by one or two
LHLHLLLL	sign/magnitude two's complement
LHHLLLLL	two's complement multiply (last cycle)
HLLLLLLL	single length normalize
HLHLLLLL	double length normalize and first divide
HHLLLLLL	two's complement divide
HHHLLLLL	two's complement correction and remainder
LLLLHHHH	arithmetic shift right F to RAM, hold Q
LLLHHHHH	logical shift right F to RAM, hold Q
LLHLHHHH	arithmetic shift right (F and Q)
LLHHHHHH	logical shift right (F and Q), F to RAM
LHLLHHHH	output F to RAM, hold Q
LHLHHHHH	output F, logical shift left Q
LHHLHHHH	output F, move Q to ALU shifter
LHHHHHHH	output F to RAM, move Q to ALU shifter
HLLLHHHH	arithmetic shift left F to RAM, hold Q
HLLHHHHH	logical shift left F to RAM, hold Q
HLHLHHHH	arithmetic shift left (F and Q), F to RAM
HLHHHHHH	logical shift left (F and Q), F to RAM
HHLLHHHH	output F, hold Q
HHLHHHHH	output F, logical shift left Q
HHHLHHHH	SIO_0 to Y_{0-3} to RAM, hold Q
HHHHHHHH	output F to RAM, hold Q

(F and Q are outputs of the ALU and Q shifters. R and S are the inputs to the ALU.)

The Am2904 is used as the status and shift control unit for the processor. This chip monitors the status of the ALU through two internal registers, the machine status register and the micro status register. The zero, carry, overflow, and sign bits are preserved by these registers.

Either, both, or neither of these internal registers can be modified with every ALU operation depending on the instruction and enable lines from the pipeline. For this design, individual microprogram operations will effect the micro status register while the machine status register will indicate the outcome of a macroinstruction. Bits in both of these registers can be modified individually or as a whole. Actual instructions (with CE_u* driven low) allow the micro status register to be modified while specific enable lines from the pipeline (CE_M*, E_C*, E_Z*, E_N*,and E_{OVR}*) allow the machine status register to be altered with one instruction. These registers are modified on the riding edge of the clock.

The I_C, I_N, I_Z, and I_{OVR} lines are connected directly to the condition code output lines of the ALU's most significant slice. These values will be latched into the desired status register with the clock or they may be directed straight to the output mux. Table 9 shows the instructions regarding loading and modifying both the micro and machine status registers.

The Y port is a three state bi-directional port from which the status registers may be written or read depending on the pipeline instruction and the enable lines. This port provides a very simple way to push and pop these registers to and from the stack in the event of an interrupt.

Table 9

COMMAND	I_{5-0} (in octal)
reset zero bit on uSR	10
set zero bit on uSR	11
reset carry bit on uSR	12
set carry bit on uSR	13
reset sign bit on uSR	14
set sign bit on uSR	15
reset overflow bit on uSR	16
set overflow bit on uSR	17
load MSR to uSR	00
set uSR and MSR	01
register swap	02
reset uSR and MSR	03
load uSR with overflow retain	06,07
load uSR with carry invert	30,31,50,51,70,71
load uSR from I_C, I_N, I_Z, I_{OVR}	4,5,20-27,32-47,52-67,72-77
load Y_C, Y_N, Y_Z, Y_{OVR} to MSR	00
invert MSR	05
load MSR from I_C, I_N, I_Z, I_{OVR}	06,07,12-27,32-47,52-67,72-77

(CE_M*, E_C*, E_Z*, E_N*, E_{OVR}*, and CE_u* must all be tied low for the full operation to take place. Any non-selected lines will not be affected.)

The Am2904 also has internal testing capabilities. All of the software branch instructions can be tested through the use of this feature. Table 10 shows the proper instruction codes to test for a series of conditions. The outcome of the test is directed

through the CT pin which is connected to the condition code mux mentioned earlier. The condition code mux and the Am29811 require an active high signal to indicate the passing of a test.

Table 10

RELATION	I_{3-0} (in hexadecimal) unsigned	signed
A = B	4	4
A <> B	5	5
A >= B	A	3
A < B	B	2
A > B	D	1
A <= B	C	0

The Am2904 also has internal logic for arithmetic shifts and rotates. This logic uses the SIO_0, SIO_n, QIO_0, QIO_n pins which are connected to most and least significant slices of the Am2903 shift input and output pins (see Diagram 2). There are thirty-two possible shifting combinations available, Table 11 illustrates some of these.

Table 11

I_{10-6}	SHIFTING FEATURE
HLLLL	shift RAM and Q left with shift-out of RAM to M_c. shift-in = 0 for both registers
HLHLL	shift RAM and Q left. shift-in of Q = 0 shift-out of Q is shift-in of RAM
LLLLL	shift RAM and Q right. shift-in of both registers is zero

Table 11 cont.

I_{10-6}	SHIFTING FEATURE
LHHHL	shift RAM and Q right. shift-in of RAM is I_N XNOR I_{OVR} and shift-in of Q is shift-out of RAM
LHLLH	rotate RAM right with carry. rotate Q right
HHLLH	rotate RAM left with carry. rotate Q left

The last device in the ALU design is the high-speed look-ahead carry generator (Am2902). This device requires only two levels of logic to produce the carries across four Am2903 slices. The only connections to this chip are through the carry-in, the carry generate and the carry propagate lines on the 2903s and the carry-in line from the Am2904.

The MAR, MBR, and Control Bus:

The memory address and the memory buffer registers connect the ALU to any external memory or I/O. Both of these buffers are three state devices and are 16-bits wide. Two Am2920s are used for the MAR. The Am2920 is essentially an octal D-type flip-flop. These devices have transparent latches and new data may be entered into the device as long as the enable line (from the pipeline) and the clock are held low. The inputs to the MAR are connected to the Y_{0-3} port of the Am2903.

The MBR uses four Am2917 bus transceivers. This transceiver allows for input and output to the data bus. Input to the Am2917

is from the Y_{0-3} port of the Am2903 while output is fed directly into DA_{3-0} port of the same device. Two control lines are used to manage this device, the bus enable BE* and the receiver latch enable RLE*. When the bus enable and the clock are active, the inputs to the transceiver are driven onto the data bus, otherwise they are high impedance. All data read from or written to the data bus is inverted from the 2903. To counteract this effect, all values are inverted within the ALU before being exported to the data bus. The receiver latch enable allows data from the bus to be passed into the ALU. When the RLE* and the clock are active, the receivers are transparent. Once the RLE* is disengaged, the data is latched.

The external control bus consist of three lines, the R/W*, IME*, and the EME. The R/W* is simply a read-write line which activates the selected device's output driver when high. All of the memory and I/O devices are connected to this pin. The IME* is a special control line active only at RESET and is exclusively connected to the reset PROM's output drivers. When this line is active, the EME line must be disabled in order to avoid bus contention. The EME line enables the external memory and the I/O modules. This is needed in order to prevent accidental memory corruption and possible bus contention with the RESET and interrupt ROM drivers.

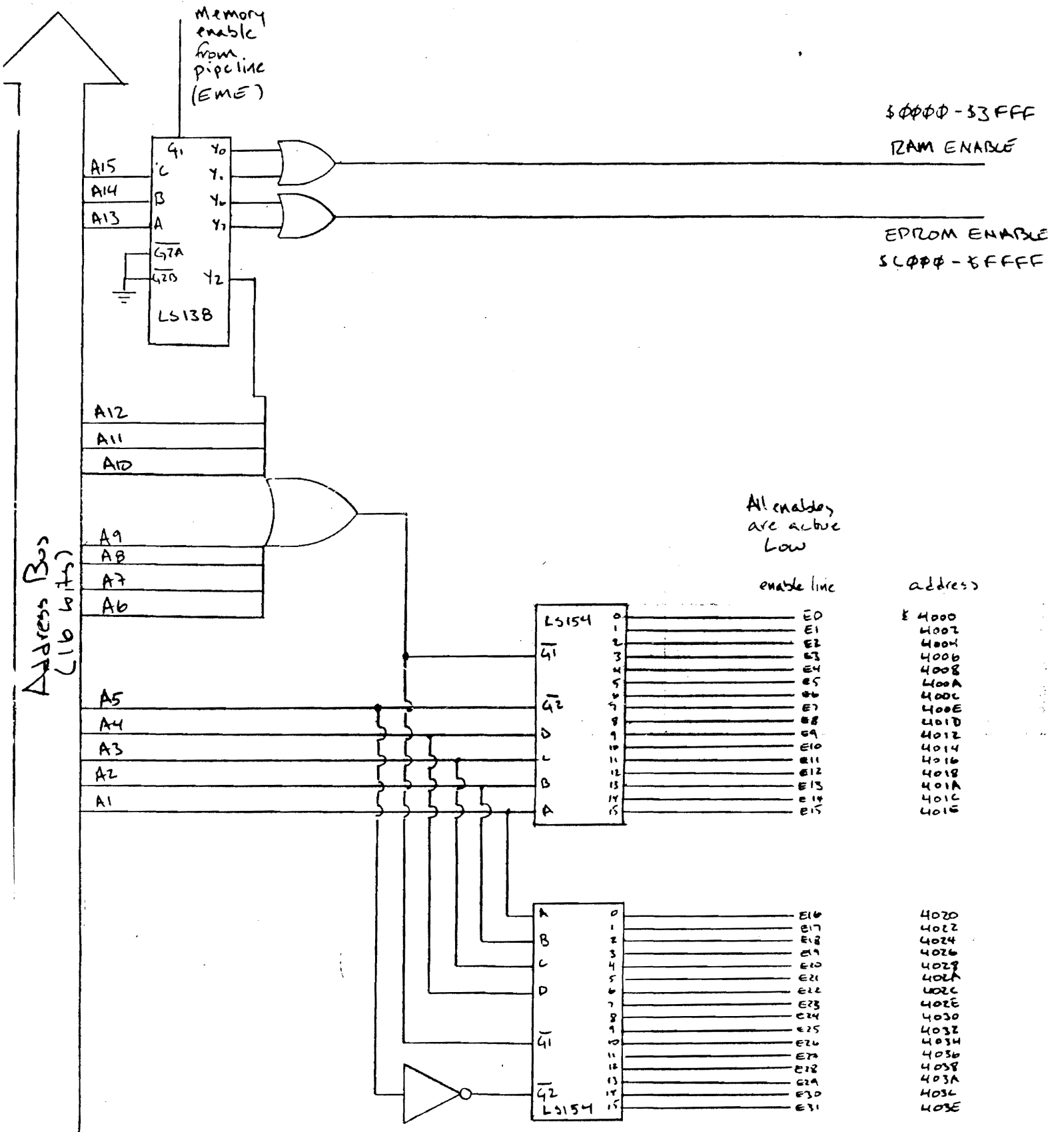
External memory:

This processor has a 64K memory space which the RAM, ROM, and I/O share. Figure 1 illustrates a memory map for this processor.

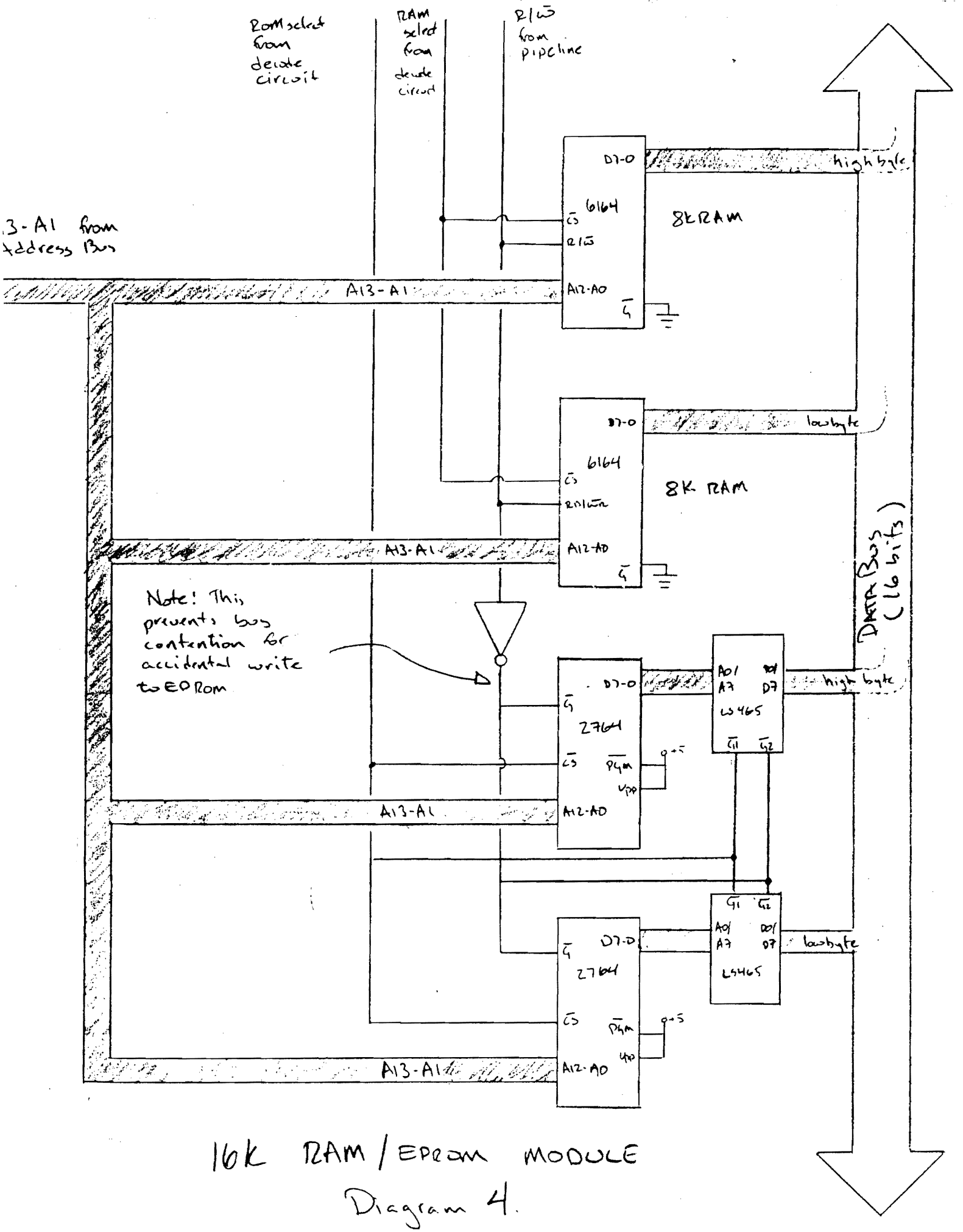
There are 8K words of RAM and EPROM memory. The RAM resides at the bottom of the memory (locations $0000 to $3FFF) while the EPROM resides at the top (locations $C000 to $FFFF). Par convention, the interrupt vectors are located at the very bottom of the RAM memory. The I/O modules are memory addressable at locations $4000 to $403E. Decoding has been provided for up to 32 modules although all of these are not used. Diagram 3 shows the decoding circuit. These enable lines become valid only if the external memory enable from the pipeline is active.

The RAM module is made of two MC6164 8K memories (see Diagram 4). When addressed, both of these units become active with the high byte being addressed by one chip and the low byte addressed by the other. Only one enable line, originating from the decode circuit in Diagram 3, is used since the address and data busses are not multiplexed. The R/W* line determines whether the output drivers or the input latches are to be used. It should be pointed out that the least significant bit is not used in the memory decode and that all memory must be addressed a word at a time beginning with an even byte.

The EPROM module is decoded in the same fashion as the RAM. The memory enable originates in the decode circuit in Diagram 3. A precaution has been added which will prevent the output drivers on the EPROM from being enabled if erroneously written. This will prevent any contention between the MBR and the EPROM. The program enable lines of the EPROM are always disabled. A set of 74LS465 bus drivers were added to the outputs of the EPROMs. Without these drivers, the memory chips would not be able to drive the data bus.



MEMORY AND I/O DECODE CIRCUIT Diagram 3.



16K RAM / EPROM MODULE

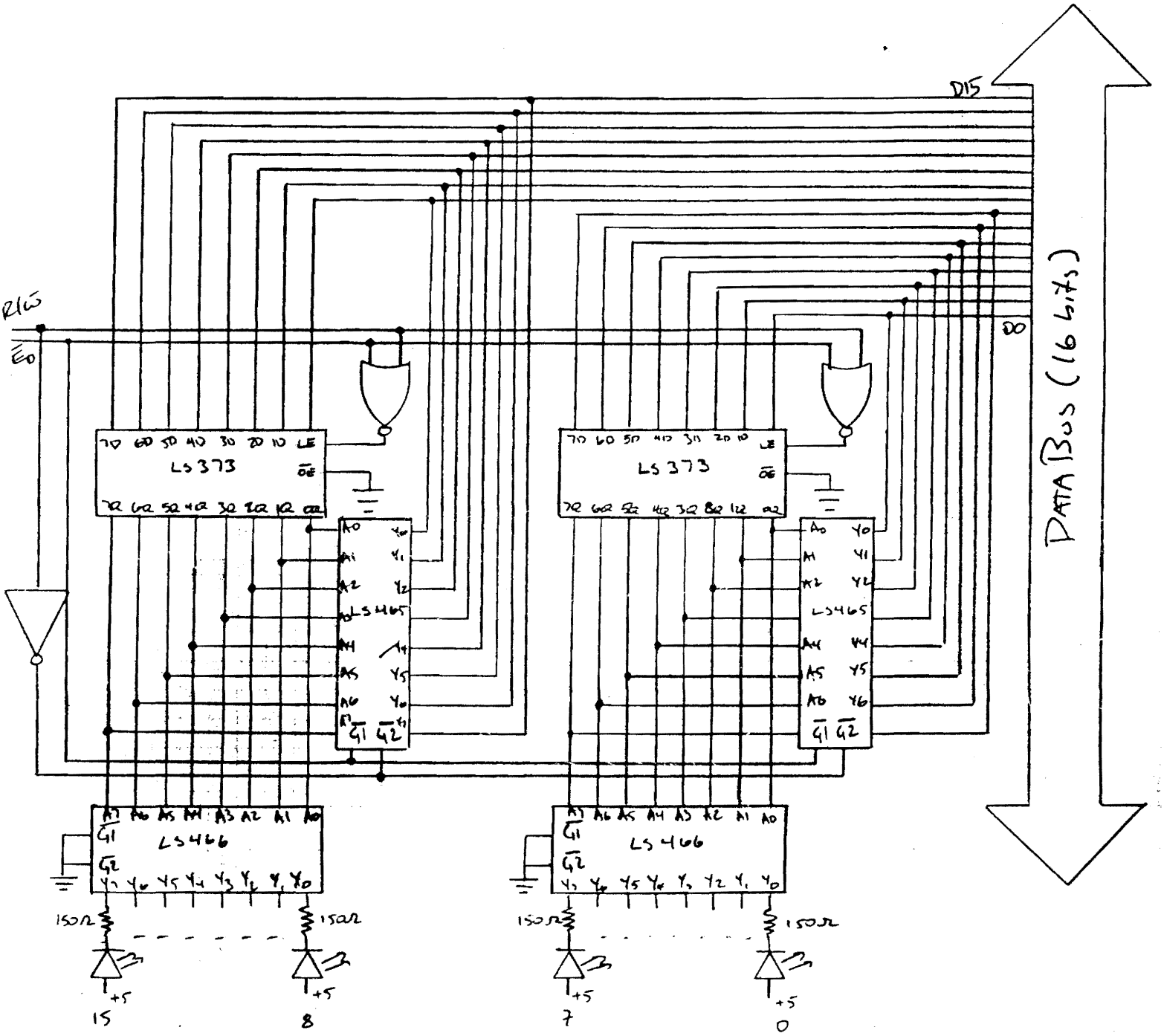
Diagram 4.

I/O Modules:

The first I/O module is the 16-bit LED display module which is shown in Diagram 5. This unit is addressed at $4000 and information regarding it can either be written or read. A series of latches and drivers are provided to allow the read-write feature. Each LED is biased at 25mA by a limiting resistor. The 74LS466 is an inverting driver capable of handling this current. The 74LS373 is a transparent latch which accepts data from the bus when the E_0* line and the R/W* line are both low. Information may be read from the unit when E_0* is low while the R/W* line is high.

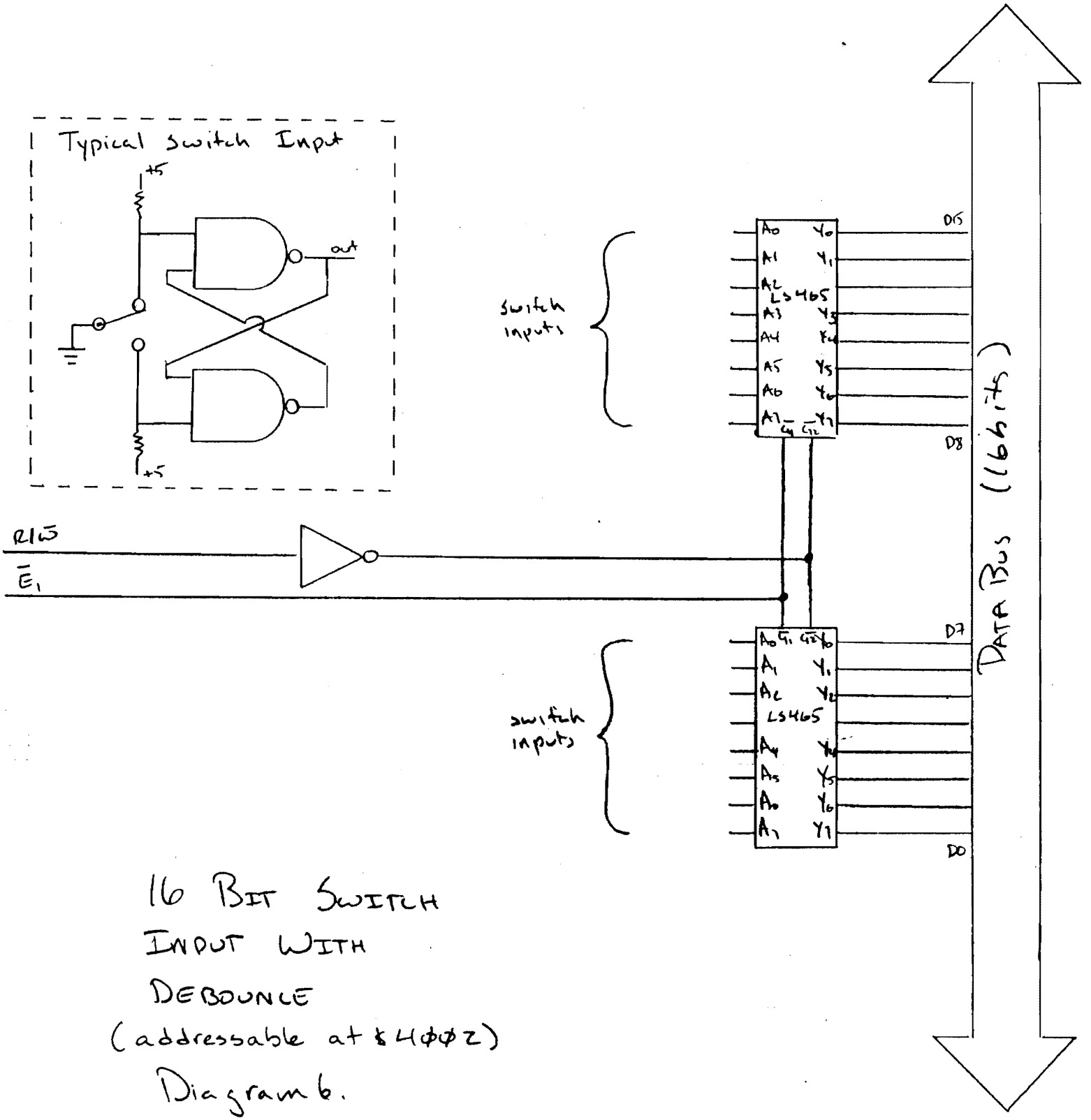
The second module is an input-only device. The 16-bit switch module is shown in Diagram 6. All of the inputs use single pole-double throw switches which employ a standard debounce circuit. This module is addressed at location $4002. The switches are connected to non-inverting drivers which become active only when the E_1* line is low while the R/W* is high.

The solid state relay module is shown in Diagram 7. This module is identical to the LED module except it is addressed at $4004 and uses the E_2* enable line. Each output of the module drives an opto-coupler which has the effect of isolating the SSR from the processor. The receivers on the opto-couplers are connected to 74LS06 devices which are high voltage open collector drivers. It is the output of the 74LS06 which actually drives the relay. A 0.1uF capacitor is added to the output to reduce AC noise.



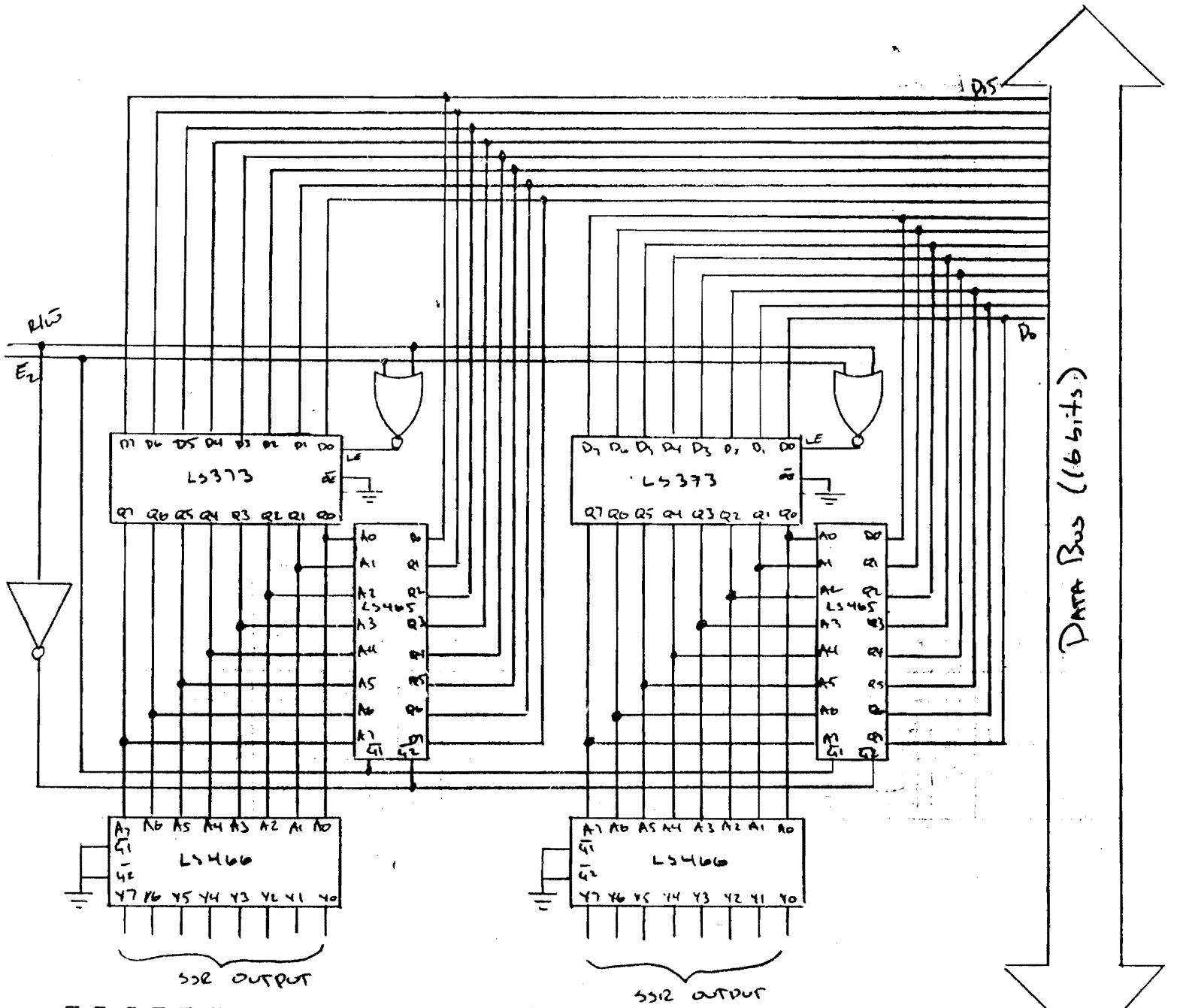
16 bit LED DISPLAY
(addressable at $4000)

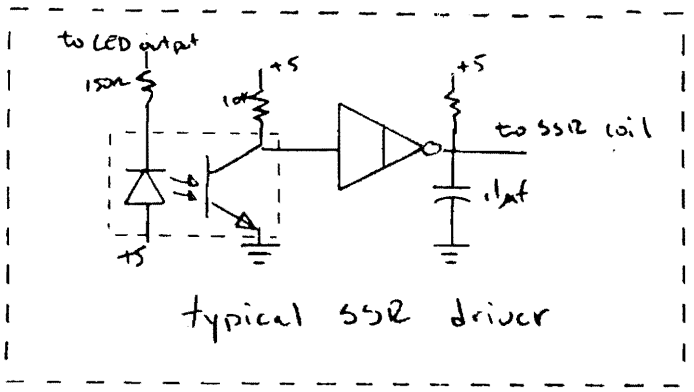
Diagram 5



16 Bit Switch Input With Debounce
(addressable at $4000)

Diagram 6.



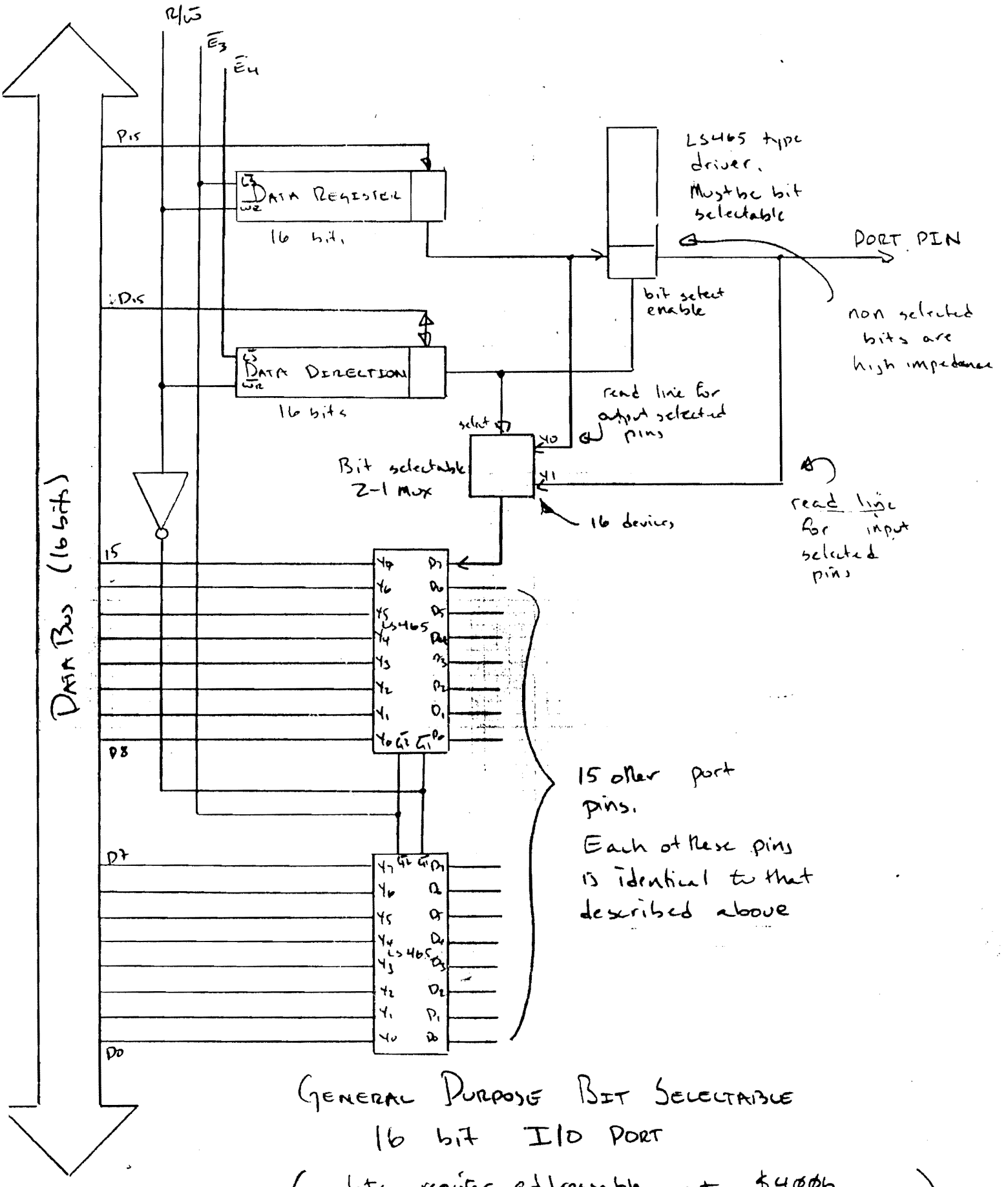


16 bit SSR
Driver
(addressable at $4004)

Diagram 7

A general purpose, 16-bit input/output port is shown in Diagram 8. Unlike the previous modules, this unit contains two registers, a data register and a data direction register. The data register is a 16-bit D-type flip-flop with a transparent latch. The data direction register is virtually identical to the setup used in the LED module. The output of the data register is driven to the output pin only if the corresponding bit in the data direction register is set. If the bit is clear, the output driver is disabled, indicating an input pin. The data register may be read just like any of the other modules, however, the origin of the information on the line depends on whether it is an input or an output. If the pin is an input, the information is read directly from the input pin, and if the bit is an output, the data is transferred directly from the data register latch. This control is dependant on the status of the data direction register. The data register is addressed at $4006 and the data direction register is located at $4008.

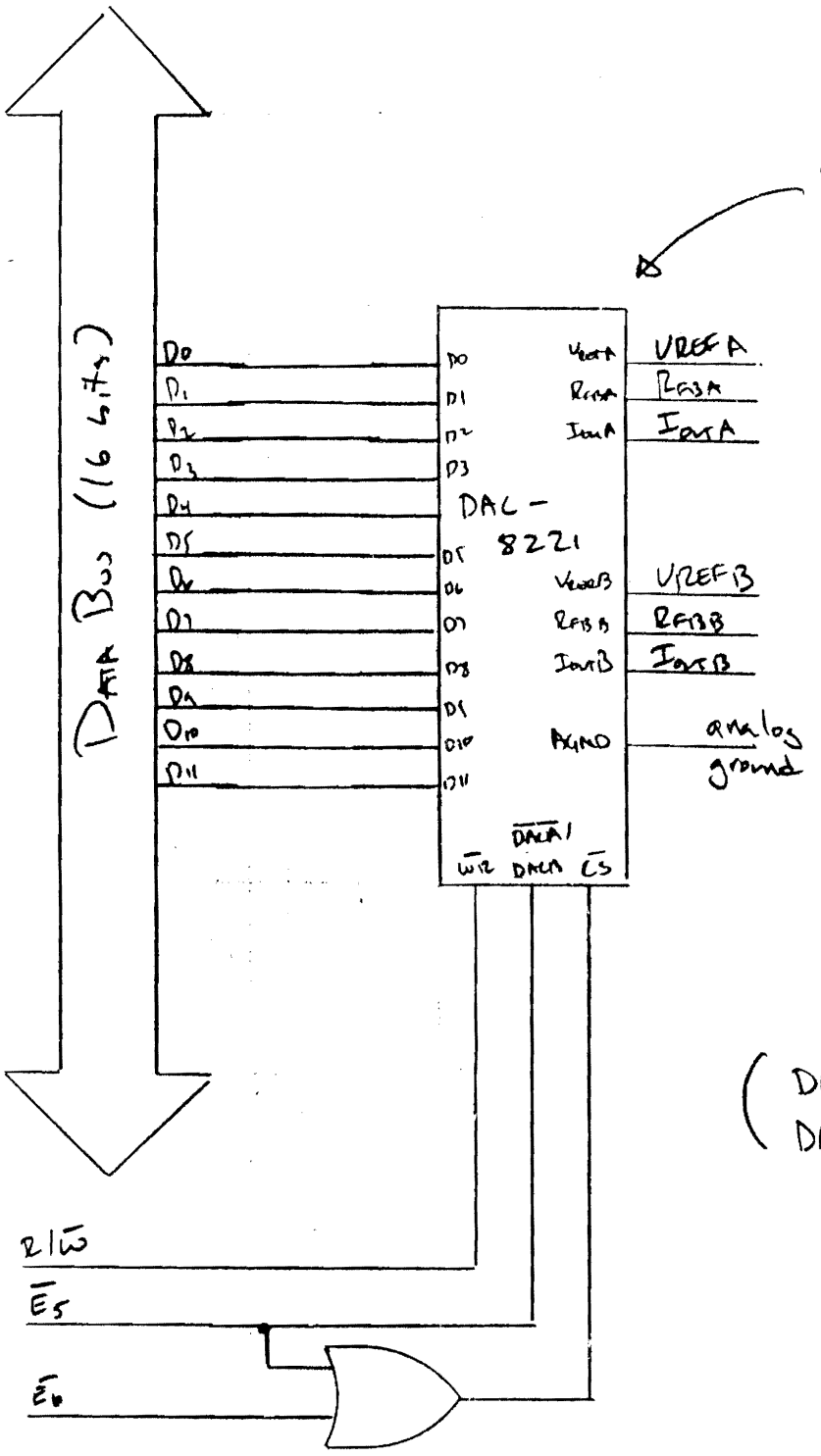
A dual 12-bit digital to analog converter is also attached to the data bus. The Precision Monolithics DAC-8221 was used to interface with the processor. DAC A is located at $400A while DAC B is at $400C. The circuit is shown in Diagram 9 and the DAC's data sheets are attached at the end of the report. This device is a write only unit. The DAC has internal transparent latches which read data from the bus whenever the chip is addressed (line E_5* or E_6*) and the R/W* line is low. External voltage references and output circuits can be attached to this device for bi- or uni-polar



GENERAL PURPOSE BIT SELECTABLE
16 bit I/O PORT

(data register addressable at $4006
data direction register addressable at $4008)

Diagram 8.



DUAL 12 BIT
D/A CONVERTERS

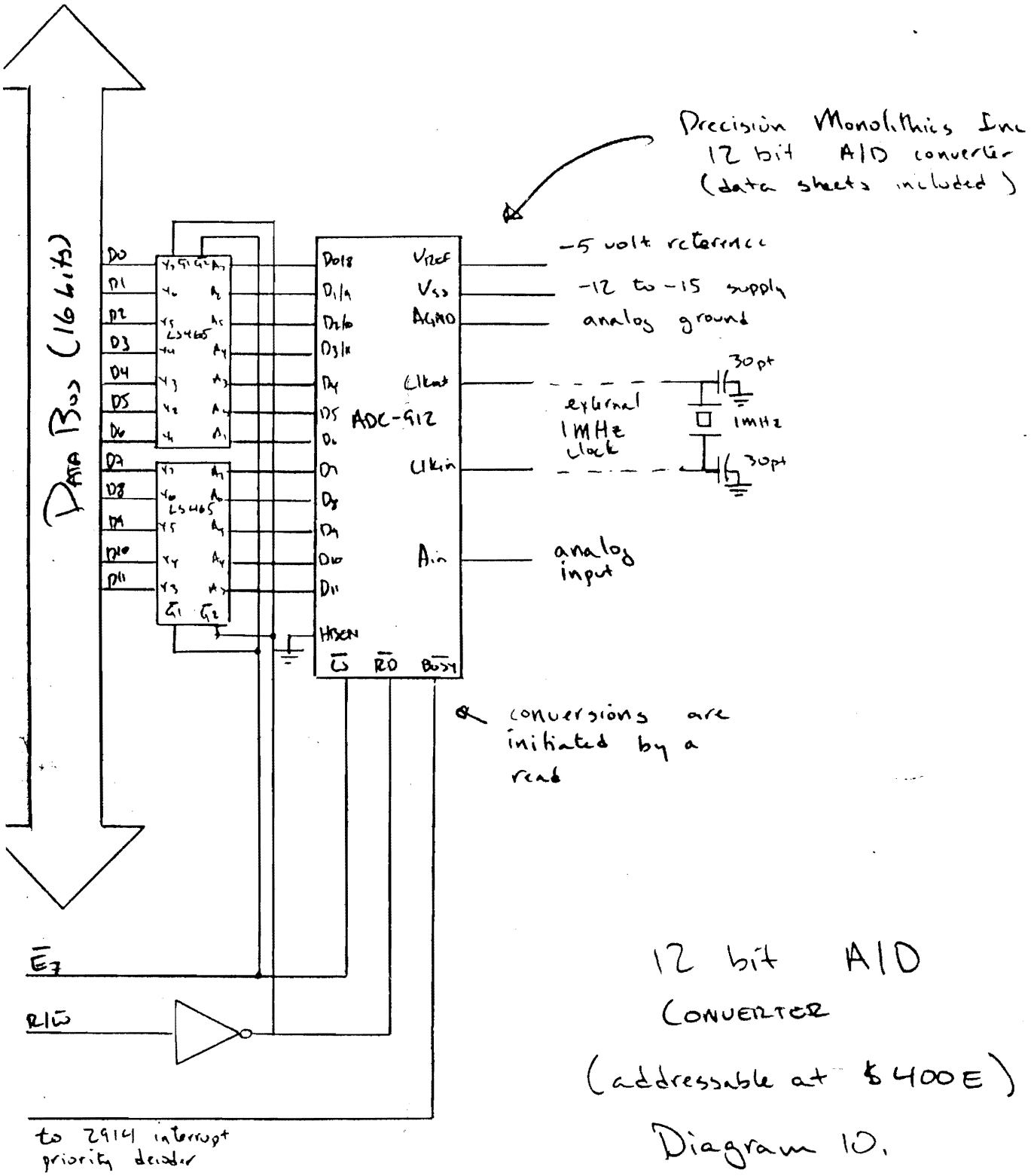
(DAC A addressable at $400A
DAC B addressable at $400C)

Diagram 9

operation. The DAC-8221 has a typical relative accuracy and a differential nonlinearity of 1 LSB.

A twelve bit analog to digital converter has also been included and is present in Diagram 10. The Precision Monolithics ADC-912 was selected for this purpose (data sheets included). Its three state output lines are connected directly to the data bus. This device is addressable at $400E and employs the E_7* enable line. The output of the last conversion is driven to the data bus whenever E_7* is active and the R/W* line is high. An external 1MHz clock is required for this device. Also, a -5V reference and a -12 to -15 volt supply are required for this chip. Unlike previous modules, this unit can create an interrupt request. The BUSY* line is connected to the priority interrupt decoder and is pulled low whenever a conversion has been requested. A conversion requires 12usec and is initiated whenever the CS* and RD* lines return high. Two 74LS465 non-inverting octal drivers were added at the outputs of the ADC to provide better bus driving capabilities.

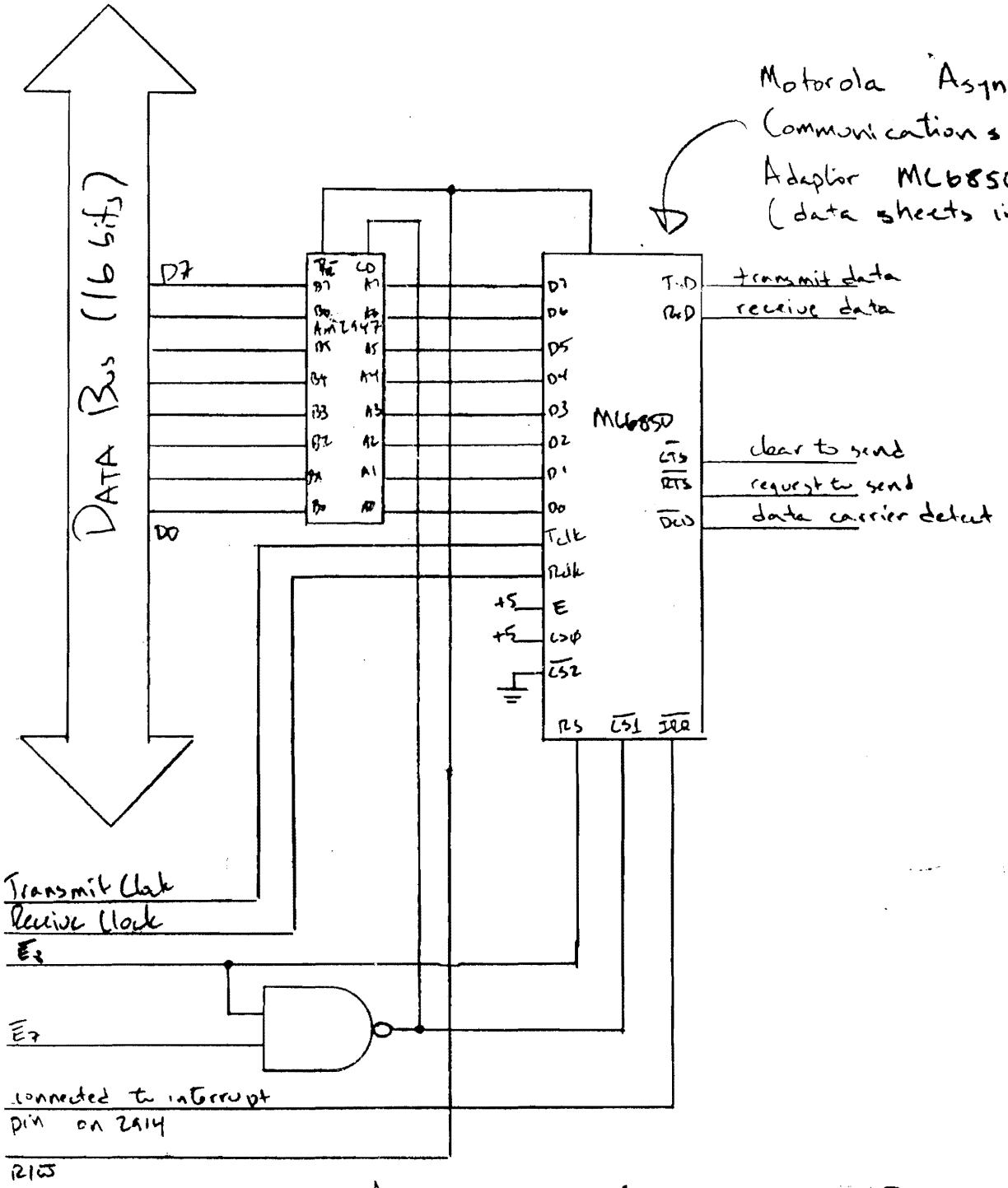
The final I/O module is the asynchronous communications interface. Diagrams 11 and 12 illustrate the design of this module. The MC6850 was selected due to its versatility and its programmability. The 6850 is capable of eight or nine bit transmission with optional even or odd parity. The device is double buffered to allow for continuous transmission and receiving. The module is programmable for 9600,4800,2400,1200,600, and 300 baud. The receiver and transmitter may operate at different



12 bit A/D CONVERTER

(addressable at $400E)

Diagram 10.



Asynchronous Communications Interface

(transmit data register (write only) addressable at $4000E
receive data register (read only) addressable at $4000E
control register (write only) addressable at $4010
status register (read only) addressable at $4010)

Diagram 11

Programmable Baud Rate Selector For Asynchronous Communications Interface (addressable at $4012)

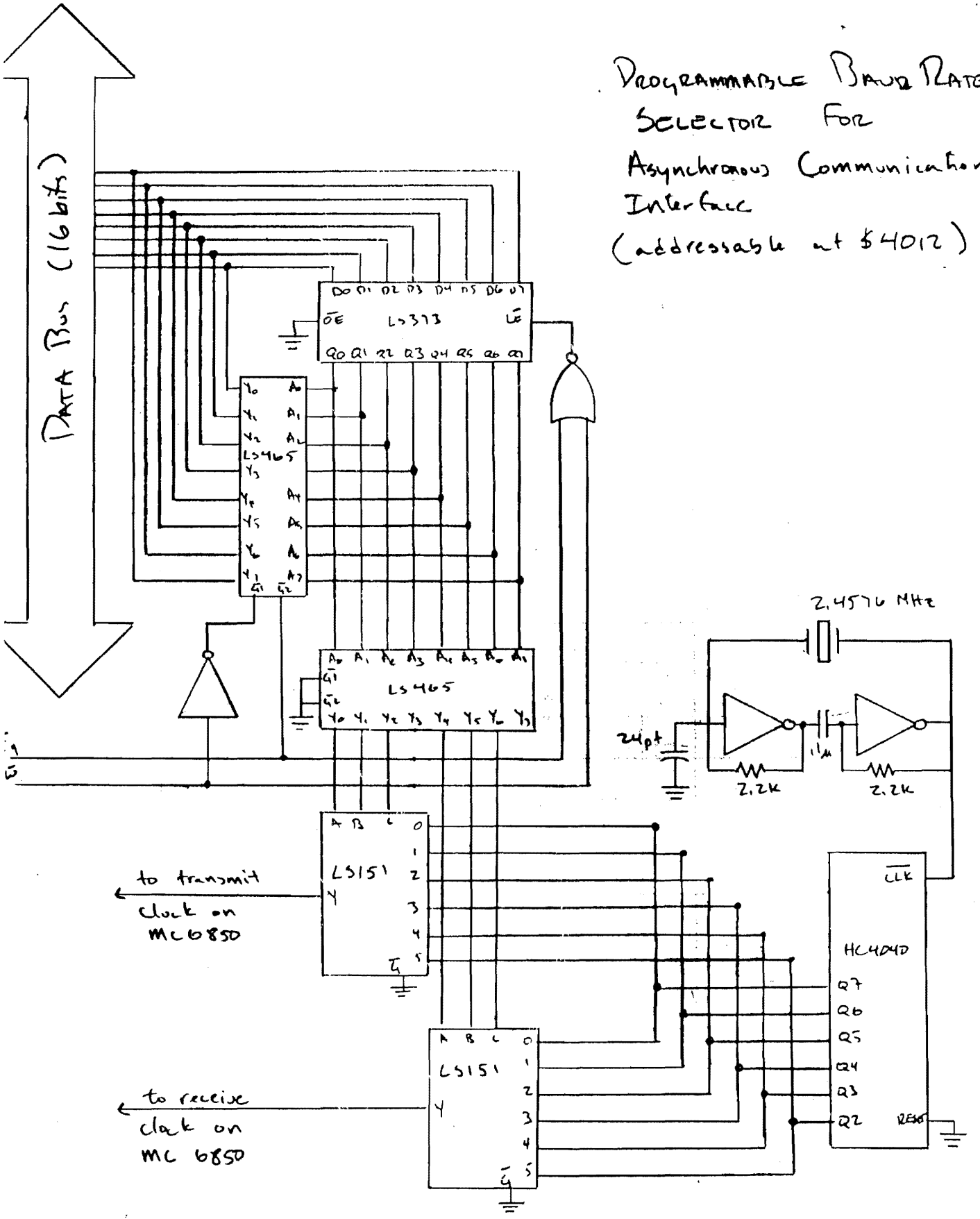


Diagram 12

rates. The baud rate selector is addressable at $4012 and the baud rate settings are shown in Table 12. The transmitter baud rate is the lowest three bits, while the receiver baud rate is in bit locations 4,5, and 6.

Table 12

A	B	C	BAUD RATE
L	L	L	9600
L	L	H	4800
L	H	L	2400
L	H	H	1200
H	L	L	600
H	L	H	300

The UART itself contains four internal registers two of which are read only while the other two are write only. The transmit/receive register is located at $400E while the control/status register is located at $4010. This module is also able to create an interrupt request. The interrupt line is attached to the priority interrupt decoder. An Am2947 bi-directional bus transceiver was connected between the UART and the data bus. The CS* and R/W* control lines of the UART determine which direction the bus will drive. The transceiver outputs are three state when not in use. This addition greatly improves the module's ability to drive the bus.

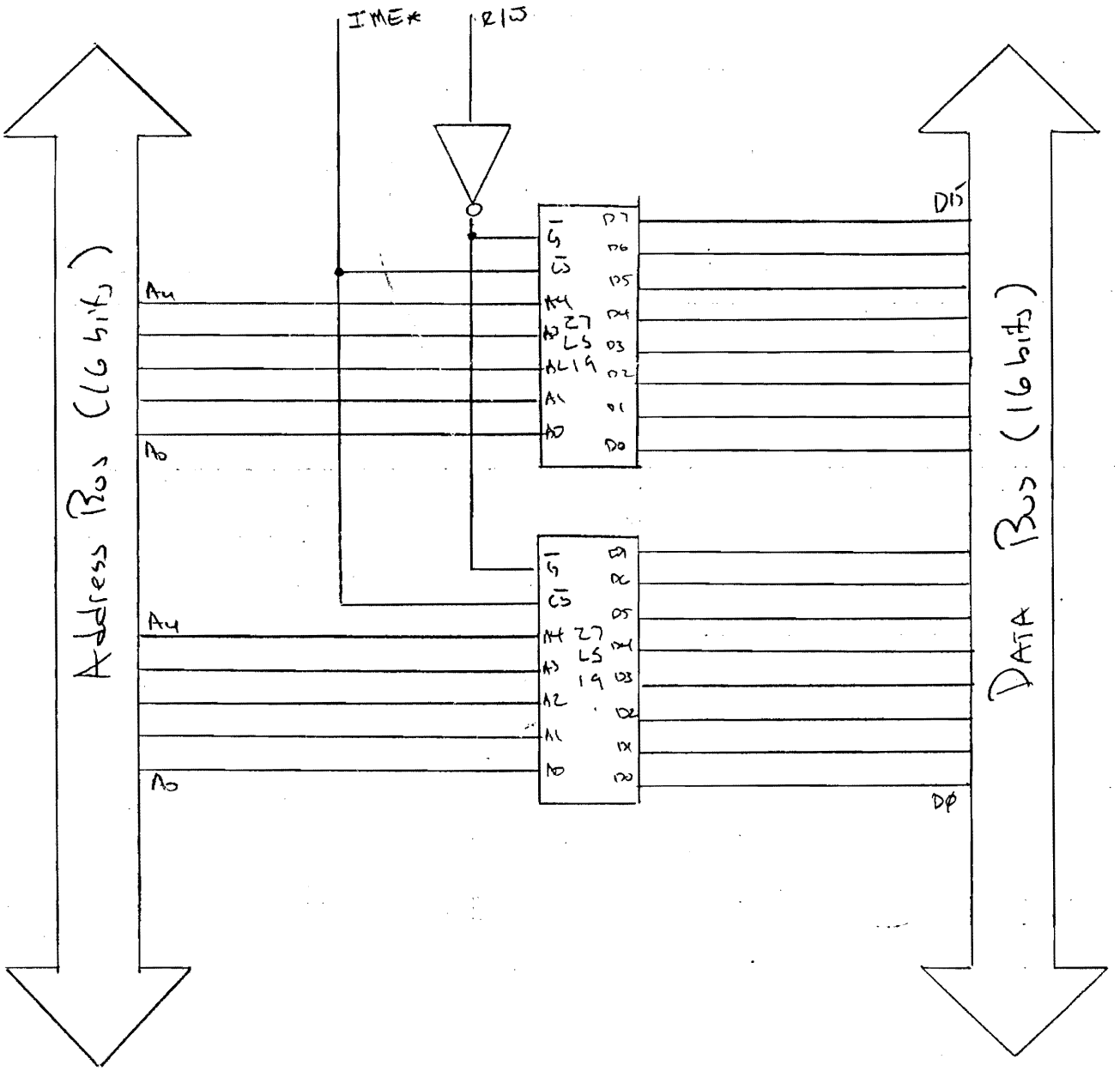
Reset:

An internal 32 word ROM has been provided for RESET. This memory contains some initial values that need to be loaded into the ALU at power-up and is shown in Diagram 13. When the processor recovers from a RESET, it will initialize itself by loading the 16 ALU RAM registers with the contents of the first 16 words of this internal ROM. By doing this, all of the constants and masks needed for operations can be completely internal to the device and the EPROM will not need to be programmed with this information. The internal ROM is only active when the IME* line is held low. By doing this, the external and internal memories will not conflict. The internal memory enable is a different polarity than the external and interrupt vector ROM enables, therefore, when the pipeline register is initially cleared, there will not be any bus contention.

The clock:

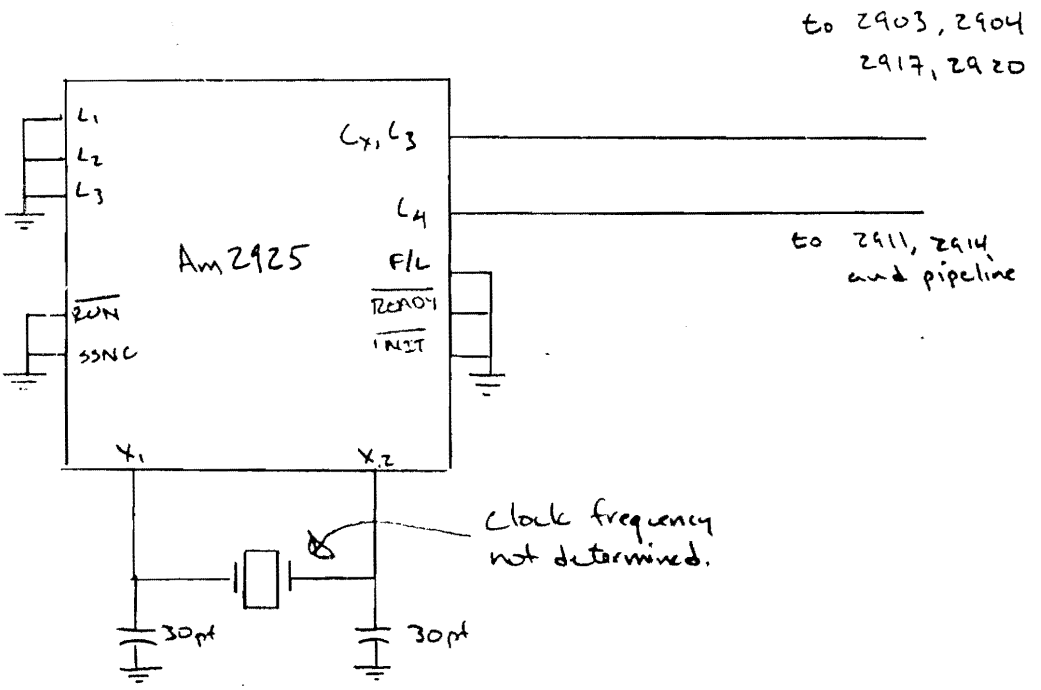
The processor utilizes an Am2925 two phased internal clock. The entire CCU operates on one cycle while the ALU, MBR, and MAR operate on the other.

The Am2911, Am2914, and the pipeline are connected to the first clock phase. Since all of these devices have transparent, active low clocks, the pipeline does not need to be clocked separately from the rest of the CCU. The CCU clock will drive high before the ALU is activated. This is essential since the next address controller is not synchronized by the clock. If the state of the 29811 were to change while the clock was low, the



32-word Internal ROM used during RESET

Diagram 13



Internal Multi-phased Clock

Diagram 14.

microprogram state may accidentally be skipped and the macroinstruction would fail.

The Am2903s, Am2904, Am2917, and the AM2920 are all connected to the second clock phase. By the time this phase becomes active, the pipeline should be set to implement the next microinstruction. The MBR and the MAR are transparent while the internal latches within the ALU and the status control register change with the low-to-high transition of the clock. This should not create any problems since this phase is also active low.

The actual clock speed is indeterminate due to the lack of information regarding the capacitance on the buses. This value would need to be determined when the circuit was realized. Finally, the clock is set in a free running mode without allowing for any wait states or internal clock halts (cycle stretches).

Bus constraints:

Some considerations have been made regarding static bus conditions. Table 13 lists every device attached to the data bus with their respective sink and source currents. The lowest source current must be greater than the sum of the high-level high-impedance leakage currents in order for the bus to operate properly. This also holds true for sink currents and low-level leakage currents.

Table 13

DEVICE	IoH	IoL	IozH	IozL
IR (74LS373)	---	---	20u	20u
MBR (Am2917)	20m	48m	50u	200u
LED module (74LS373)	---	---	20u	20u
(74LS465)	2.6m	24m	20u	20u
Switch module (74LS465)	2.6m	24m	20u	20u
ADC module (74LS465)	2.6m	24m	20u	20u
DAC module (DAC-8221)	---	---	10u	10u
UART module (Am2947)	10m	40m	200u	200u
SSR module (74LS373)	---	---	20u	20u
(74LS465)	2.6m	24m	20u	20u
Am2914 M bus drivers	2.6m	12m	150u	150u
RAM module (MC6164)	4m	8m	2u	2u
EPROM module (74LS465)	2.6m	24m	20u	20u
general I/O port (74LS373)	---	---	20u	20u
(74LS465)	2.6m	24m	20u	20u
baud rate select (74LS373)	---	---	20u	20u
(74LS465)	2.6m	24m	20u	20u
TOTAL			632u	782u

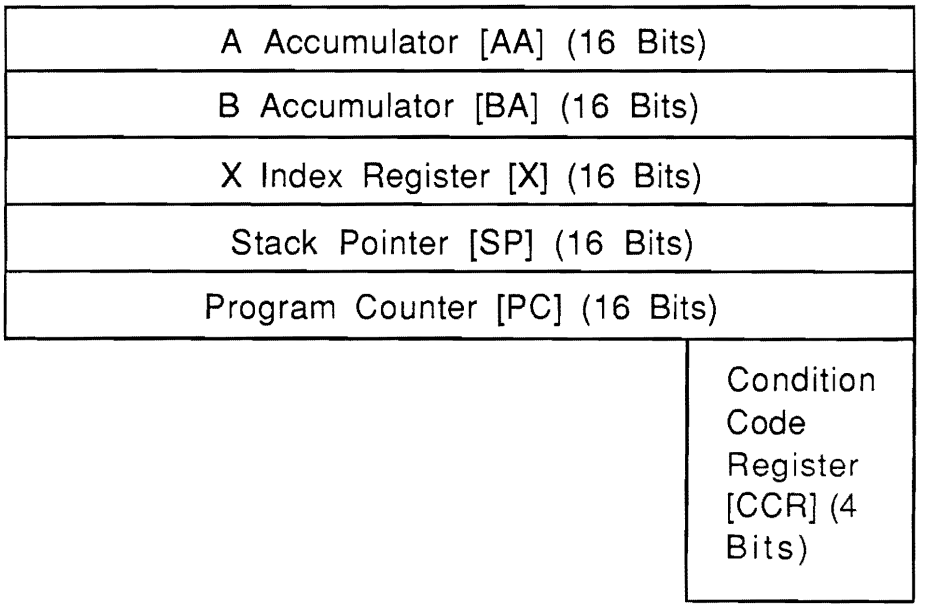
From the given information, it can be seen that the smallest source and sink currents are 4 and 8 milliamps while the largest leakage currents are 632 and 782 microamps. This indicates that the bus is not overloaded and has plenty of room for additional components.

The noise immunity can be calculated by finding the driver voltages at the rated output currents. For the high state, the lowest V_{OH} value is 2.4 volts without any hysteresis. Since the TTL high logic level is 2.0 volts, the noise immunity is 0.4 volts. Similarly, the highest V_{OL} voltage is 0.45 volts which yields a low-level noise immunity of 0.35 volts.

CPU - PROGRAMMER'S MODEL

This section discusses the register configuration seen at the macro programming level.

In this design, the registers are integral to the CPU and are not memory addressable. There are two 16-bit accumulators available for mathematical and logical operations. One index register has been provided for indexed addressing operations. The microprogram uses some of the 64K memory for stack operations, so a stack pointer is used to point at the address of the next free word on the stack. The program counter points at the next opcode to be implemented. The last user accessible register is the condition code register which shows the flags resulting from arithmetic and other operations as they occur in the ALU. These four flags are updated during every instruction performed by the CPU.



Accumulators (A and B) :
Accumulators A and B are general purpose 16-bit registers available for all mathematical and logical functions performed by the CPU. Although most operations can be performed by either register, some are register specific (TSB, TBS, and ABX).

Index register (X) :
The 16-bit index register X is primarily designed for indexed addressing. In this mode, an 8-bit offset (unsigned) is temporarily added to the 16-bit X register to form the effective address of the operand for an instruction.

Stack Pointer :
The Stack Pointer is a register that contains the address of the next free word on the stack. Most applications will require that this register be loaded with the address of a free region of RAM in the 64K memory space.

When an operation requires that a word be added to the stack, the stack pointer provides the address of the location to place the word. After the word is stored, the SP is automatically decremented to point at the next available free space. Likewise, when an operation needs a word from the stack, the SP is automatically incremented and the last word placed on the stack is read and placed in the appropriate register. Note that all operations use word rather than byte format, so all increments and decrements are by 2.

Some operations including WAI and JSR require that the stack be used to store various registers. The companion operations RTI and RTS require the stack to retrieve the original register values.

An interrupt will cause all registers to be placed on the stack and the program register to be loaded with the interrupt routine's location. After completing the interrupt service routine, all registers will be replaced with their original values and the program will resume.

Program Counter (PC) :
This is a 16-bit register that contains the address of the next operation to be executed. This register can be modified by a branch or jump instruction. An interrupt will also cause a temporary modification to the PC.

Condition Code Register (CCR)
This is an 4-bit register that contains the four flags generated by the ALU after arithmetic and other operations have been performed. The flags of this register are updated after each instruction. Certain branch instructions, as well as many arithmetic and logical instructions, use these flags in their operations. Two of these flags can be set or reset with specific instructions (carry and overflow). The other two flags (zero and negative) can only be modified by performing an operation that generates either of the two conditions.

ADDRESSING MODES

This section describes each available addressing mode used in this CPU. The available modes are : Immediate (IMM), Direct (DIR), Indirect (INDIR), Extended (EXT), Indexed (INDX), Inherent (INH), and Relative (REL). Only some of the addressing modes can be used with each instruction. See the individual instruction definition or examine the Valid Addressing Mode Table for available modes.

The following symbols are defined :

$	=	Hex
@	=	Octal
%	=	Binary
'	=	Single Ascii Character
None	=	Decimal
AA	=	A Accumulator
BA	=	B Accumulator
X	=	X Register
op	=	Opcode (1 byte)
hh	=	High order byte
l l	=	Low order byte
co	=	2's complement offset

Immediate Addressing (IMM) :

General format : op 00 hh ll

*1 byte Immediate format : op ll

In immediate addressing mode, the two byte immediate operand is contained in the 2 bytes following the opcode word. The hex values of the operand are split into high byte and low byte are stored in this order immediately following the opcode word.

Only 1 instruction uses the 1 byte immediate format (See * above). This format allows the user to set or reset any of the Interrupt mask bits. In this format, the byte immediately following the opcode byte contains the 8 bit mask that will be used to replace the old mask.

Example 1:

Instruction - Place $1000 into the A accumulator using immediate addressing.

Uncompiled line : LDA #$1000

Compiled line : 5D 00 10 00

Result : AA = $1000

Example 2:

Instruction - Replace current Interrupt Mask with $15

Uncompiled line : SMSK $15

Compiled line : E9 15

Result : MASK = 15

Direct Addressing (DIR) :
General format : op ll

This addressing mode allows the user to address the first 256 bytes of space in the 64K of memory using one opcode word. This saves several cycles of CPU time and would be useful in an I/O intensive application. The addressing mode allows the byte immediately following the opcode to be used as this direct address.

Example 3 :
Instruction - Place the contents of memory location $30 into the A accumulator using direct addressing.
Locations $0030-$0031 contain the word $1000
Uncompiled line : LDA $30
Compiled line : 5E 30
Result : AA = $1000

Indirect Addressing :
General format : op 00 hh ll

This method of addressing uses the two bytes immediately following the opcode word as a pointer to a memory location containing the address location of the operand. The pointer is split into a high address byte and low address byte and is stored in that order.

Example 4:
Instruction - Place the contents of memory location $C030 into the A accumulator using indirect addressing.
Location $1234-$1235 contain the word $C030
Location $C030-$C031 contain the word $1000
Uncompiled line : LDA $1234
Compiled line : 5F 00 12 34
Result : AA = $1000

Extended Addressing :
General format : op 00 hh ll

Extended addressing expects the address of the operand to be in the word directly following the opcode word. This address is split into high and low bytes which are stored in that order.

Example 5:
Instruction - Place the contents of memory location $1230 into the A accumulator using direct addressing.
Locations $1230-$1231 contain the word $1000
Uncompiled line : LDA $1230
Compiled line : 60 00 12 30
Result : AA = $1000

Indexed Addressing :
General format : op ll

Indexed addressing uses the value contained in the X register as the base address and the value immediately following the opcode byte as the offset. The offset is an unsigned, single byte value.

Example 6 :
Instruction - Place the contents of memory location $4320 into the A accumulator using indexed addressing.
X register contains $4320
Locations $1230-$1231 contain the word $1000
Uncompiled line : LDA $20,X
Compiled line : 61 20
Result : AA = $1000

Inherent Addressing :
General format : op 00

Inherent Addressing means that the operands for this instruction are already contained in the CPU and are inherent to the instruction.

Example 7:
Instruction - Add the contents of accumulator A with accumulator B and place the result in accumulator A
AA = $1234
BA = $ 4321
Uncompiled line : ABA
Compiled line : 02 00
Result : AA = $5555
BA = $4321

Relative Addressing :
General format : op co

This addressing mode is used with the branch instructions to provide the ability to branch forward or backward from the current PC position. Note that the PC is automatically incremented with each instruction so a brach of 0 will not effect the programs progression, however a branch of $FE will cause the current branch instruction to be performed.

Example 8 :
Instruction - Branch forward by 6 bytes if carry bit (in CPU) is clear
PC contains $2000 immediately before this command is issued
Uncompiled line :
Compiled line : CB 06
Result : if carry is clear then PC = $2008
if carry is set then PC = $2002

INTEGER ARITHMETIC FUNCTIONS :

Valid Addressing modes with opcodes

Mnemonic	Definition	IMM	DIR	INDIR	EXT	INDX	INH
ABA	Add BA to AA						02
ABX	Add BA to X						03
ADCA	Add operand to AA with Carry	0E	0F	10	11	12	
ADCB	Add operand to BA with Carry	13	14	15	16	17	
ADDA	Add operand to AA	04	05	06	07	08	
ADDB	Add operand to BA	09	0A	0B	0C	0D	
CBA	Compare BA to AA						18
CMPA	Compare operand to AA	19	1A	1B	1C	1D	
CMPB	Compare operand to BA	1E	1F	20	21	22	
CPX	Compare operand to X	23	24	25	26	27	
DEC	Decrement operand by 1	2A	2B	2C	2D	2E	
DECA	Decrement AA by 1						28
DECB	Decrement BA by 1						29
DES	Decrement SP by 1						2F
DEX	Decrement X by 1						30
INC	Increment operand by 1	33	34	35	36	37	
INCA	Increment AA by 1						31
INCB	Increment BA by 1						32
INS	Increment SP by 1						38
INX	Increment X by 1						39
NEG	Replace operand with 2's complement of operand	3C	3D	3E	3F	40	
NEGA	Replace AA with 2's complement of AA						3A
NEGB	Replace BA with 2's complement of BA						3B
SBA	Subtract BA from AA						42
SBCA	Subtract operand from AA with carry	43	44	45	46	47	
SBCB	Subtract operand from BA with carry	48	49	4A	4B	4C	
SBX	Subtract operand from X register						41
SUBA	Subtract operand from AA	4D	4E	4F	50	51	
SUBB	Subtract operand from BA	52	53	54	55	56	

LOADS, STORES AND TRANSFERS :

Valid Addressing modes with opcodes

Mnemonic	Definition	IMM	DIR	INDIR	EXT	INDX	INH
CLR	Replace Operand with zero		57	58	59	5A	
CLRA	Replace AA with zero						5B
CLRB	Replace BA with zero						5C
LDA	Load AA with operand	5D	5E	5F	60	61	
LDB	Load BA with operand	62	63	64	65	66	
LDS	Load SP with operand	67	68	69	6A	6B	
LDX	Load X register with operand	6C	6D	6E	6F	70	
PSHA	Push AA on stack						71
PSHB	Push BA on stack						72
PSHX	Push X register on stack						73
PULA	Pull AA from stack						74
PULB	Pull BA from stack						75
PULX	Pull X register from stack						76
STA	Store AA in memory		77	78	79	7A	
STB	Store BA in memory		7B	7C	7D	7E	
STX	Store X in memory		7F	80	81	82	
TAB	Transfer AA to BA						83
TBA	Transfer BA to AA						84
TBS	Transfer BA to SP						85
TSB	Transfer SP to BA						86

LOGICAL OPERATIONS :

Valid Addressing modes with opcodes

Mnemonic	Definition	IMM	DIR	INDIR	EXT	INDX	INH
ANDA	Logical AND AA with operand	87	88	89	8A	8B	
ANDB	Logical AND BA with operand	8C	8D	8E	8F	90	
BITA	Bit test AA with operand	91	92	93	94	95	
BITB	Bit test BA with operand	96	97	98	99	9A	
COM	Replace operand with 1's complement of operand		9B	9C	9D	9E	
COMA	Replace AA with 1's complement of AA						9F
COMB	Replace BA with 1's complement of BA						A0
EORA	Logical EXCLUSIVE OR AA with operand	A1	EB	EC	ED	EE	
EORB	Logical EXCLUSIVE OR BA with operand	A2	EF	F0	F1	F2	
ORA	Logical OR AA with operand	B3	B4	B5	B6	B7	
ORB	Logical OR BA with operand	B8	A3	A4	DA	33	

INTEGER MULTIPLY AND DIVIDE :

Valid Addressing modes with opcodes

Mnemonic	Definition	IMM	DIR	INDIR	EXT	INDX	INH
IDIV	Divide signed integer in AA with signed integer in BA						A5
MUL	Multiply signed integer in AA with signed integer in BA						A6

SHIFTS AND ROTATES :

Valid Addressing modes with opcodes

Mnemonic	Definition	IMM	DIR	INDIR	EXT	INDX	INH
ASL/LSL	Perform Shift Left on bits of operand		A7	A8	A9	AA	
ASLA/LSLA	Perform Shift Left on bits of AA						AB
ASLB/LSLB	Perform Shift Left on bits of BA						AC
ASR	Perform Arithmatic Shift Right on bits of operand		AD	AE	AF	B0	
ASRA	Perform Arithmatic Shift Right on bits of AA						B1
ASRB	Perform Arithmatic Shift Right on bits of BA						B2
LSR	Perform Logical Shift Right on bits of operand		B9	BA	BB	BC	
LSRA	Perform Logical Shift Right on bits of AA						BD
LSRB	Perform Logical Shift Right on bits of BA						BE
ROL	Perform Rotate Left through carry on operand		BF	C0	C1	C2	
ROLA	Perform Rotate Left through carry on AA						C3
ROLB	Perform Rotate Left through carry on BA						C4
ROR	Perform Rotate Right through carry on operand		C5	C6	C7	C8	
RORA	Perform Rotate Right through carry on AA						C9
RORB	Perform Rotate Right through carry on BA						CA

BRANCH AND JUMP OPERATIONS :

Valid Addressing modes with opcodes

Mnemonic	Definition	IMM	DIR	INDIR	EXT	INDX	INH	REL
BCC/BHS	Branch if Carry Clear/ Greater than or equal (Unsigned numbers)							CB
BCS/BLO	Branch if Carry Set/Less than (Unsigned numbers)							CC
BEQ	Branch if Equal							CD
BGE	Branch if Greater than or Equal (Signed numbers)							CE
BGT	Branch if Greater than (Signed numbers)							CF
BHI	Branch if Greater than (Unsigned numbers)							D0
BLE	Branch if Less than or Equal (Signed numbers)							D1
BLS	Branch if Less than or Equal (Unsigned numbers)							D2
BLT	Branch if Less than (Signed numbers)							D3
BMI	Branch if Minus							D4
BNE	Branch if Not Equal							D5
BPL	Branch if Positive							D6
BRA	Branch Always							D7
BVC	Branch if Overflow Clear							D8
BVS	Branch if Overflow Set							D9
NOP	No Operation						01	
JMP	Jump to Location			DB	DC			
JSR	Jump to Subroutine			DD	DE			
RTI	Return from Interrupt						DF	
RTS	Return from Subroutine						E0	
WAI	Wait for Interrupt						E1	

CCR, INTERRUPT AND DIAGNOSTIC COMMANDS:

Valid Addressing modes with opcodes

Mnemonic	Definition	IMM	DIR	INDIR	EXT	INDX	INH
CLC	Clear Carry bit in CCR						E2
CLV	Clear Overflow bit in CCR						E4
CMSK	Clear Interrupt Mask						E8
DSI	Disable Interupts						E6
ENI	Enable Interupts						E3
SEC	Set Carry bit in CCR						E5
SEV	Set Overflow bit in CCR						E7
SMSK	Set Interupt Mask	E9*					
TEST	Test ALU and RAM						EA

* Uses 1 Byte Immediate addressing (See Programmers Model of CCU)

		Valid Addressing modes with opcodes						
Mnemonic	Definition	IMM	DIR	INDIR	EXT	INDX	INH	REL
ABA	Add BA to AA						02	
ABX	Add BA to X						03	
ADCA	Add operand to AA with Carry	0E	0F	10	11	12		
ADCB	Add operand to BA with Carry	13	14	15	16	17		
ADDA	Add operand to AA	04	05	06	07	08		
ADDB	Add operand to BA	09	0A	0B	0C	0D		
ANDA	Logical AND AA with operand	87	88	89	8A	8B		
ANDB	Logical AND BA with operand	8C	8D	8E	8F	90		
ASL/LSL	Perform Shift Left on bits of operand		A7	A8	A9	AA		
ASLA/LSLA	Perform Shift Left on bits of AA						AB	
ASLB/ASLB	Perform Shift Left on bits of BA						AC	
ASR	Perform Arithmatic Shift Right on bits of operand		AD	AE	AF	B0		
ASRA	Perform Arithmatic Shift Right on bits of AA						B1	
ASRB	Perform Arithmatic Shift Right on bits of BA						B2	
BCC/BHS	Branch if Carry Clear/ Greater than or equal (Unsigned numbers)							CB
BCS/BLO	Branch if Carry Set/Less than (Unsigned numbers)							CC
BEQ	Branch if Equal							CD
BGE	Branch if Greater than or Equal (Signed numbers)							CE
BGT	Branch if Greater than (Signed numbers)							CF
BHI	Branch if Greater than (Unsigned numbers)							D0
BITA	Bit test AA with operand	91	92	93	94	95		
BITB	Bit test BA with operand	96	97	98	99	9A		
BLE	Branch if Less than or Equal (Signed numbers)							D1
BLS	Branch if Less than or Equal (Unsigned numbers)							D2
BLT	Branch if Less than (Signed numbers)							D3
BMI	Branch if Minus							D4
BNE	Branch if Not Equal							D5
BPL	Branch if Positive							D6
BRA	Branch Always							D7
BVC	Branch if Overflow Clear							D8
BVS	Branch if Overflow Set							D9

		Valid Addressing modes with opcodes						
Mnemonic	Definition	IMM	DIR	INDIR	EXT	INDX	INH	REL
CBA	Compare BA to AA						18	
CLC	Clear Carry bit in CCR						E2	
CLR	Replace Operand with zero		57	58	59	5A		
CLRA	Replace AA with zero						5B	
CLRB	Replace BA with zero						5C	
CLV	Clear Overflow bit in CCR						E4	
CMPA	Compare operand to AA	19	1A	1B	1C	1D		
CMPB	Compare operand to BA	1E	1F	20	21	22		
CMSK	Clear Interrupt Mask						E8	
COM	Replace operand with 1's complement of operand		9B	9C	9D	9E		
COMA	Replace AA with 1's complement of AA						9F	
COMB	Replace BA with 1's complement of BA						A0	
CPX	Compare operand to X	23	24	25	26	27		
DEC	Decrement operand by 1	2A	2B	2C	2D	2E		
DECA	Decrement AA by 1						28	
DECB	Decrement BA by 1						29	
DES	Decrement SP by 1						2F	
DEX	Decrement X by 1						30	
DSI	Disable Interupts						E6	
ENI	Enable Interupts						E3	
EORA	Logical EXCLUSIVE OR AA with operand	A1	BB	BC	BD	BE		
EORB	Logical EXCLUSIVE OR BA with operand	A2	BF	F0	F1	F2		
FPAD	Add FP number in AA to FP number in BA						F3	
FPSB	Subtract FP number in BA from FP number in AA						F4	
IDIV	Divide signed integer in AA with signed integer in BA						A5	
INC	Increment operand by 1	33	34	35	36	37		
INCA	Increment AA by 1						31	
INCB	Increment BA by 1						32	
INS	Increment SP by 1						38	
INX	Increment X by 1						39	
JMP	Jump to Location			DB	DC			

		Valid Addressing modes with opcodes						
Mnemonic	Definition	IMM	DIR	INDIR	EXT	INDX	INH	REL
JSR	Jump to Subroutine			DD	DE			
LDA	Load AA with operand	5D	5E	5F	60	61		
LDB	Load BA with operand	62	63	64	65	66		
LDS	Load SP with operand	67	68	69	6A	6B		
LDX	Load X register with operand	6C	6D	6E	6F	70		
LSR	Perform Logical Shift Right on bits of operand		B9	BA	BB	BC		
LSRA	Perform Logical Shift Right on bits of AA						BD	
LSRB	Perform Logical Shift Right on bits of BA						BE	
MUL	Multiply signed integer in AA with signed integer in BA						A6	
NEG	Replace operand with 2's complement of operand	3C	3D	3E	3F	40		
NEGA	Replace AA with 2's complement of AA						3A	
NEGB	Replace BA with 2's complement of BA						3B	
NOP	No Operation						01	
ORA	Logical OR AA with operand	B3	B4	B5	B6	B7		
ORB	Logical OR BA with operand	B8	A3	A4	DA	33		
PSHA	Push AA on stack						71	
PSHB	Push BA on stack						72	
PSHX	Push X register on stack						73	
PULA	Pull AA from stack						74	
PULB	Pull BA from stack						75	
PULX	Pull X register from stack						76	
ROL	Perform Rotate Left through carry on operand		BF	C0	C1	C2		
ROLA	Perform Rotate Left through carry on AA						C3	
ROLB	Perform Rotate Left through carry on BA						C4	
ROR	Perform Rotate Right through carry on operand		C5	C6	C7	C8		
RORA	Perform Rotate Right through carry on AA						C9	
RORB	Perform Rotate Right through carry on BA						CA	
RTI	Return from Interrupt						DF	
RTS	Return from Subroutine						E0	
SBA	Subtract BA from AA						42	
SBCA	Subtract operand from AA with carry	43	44	45	46	47		

		Valid Addressing modes with opcodes						
Mnemonic	Definition	IMM	DIR	INDIR	EXT	INDX	INH	REL
SBCB	Subtract operand from BA with carry	48	49	4A	4B	4C		
SBX	Subtract operand from X register						41	
SEC	Set Carry bit in CCR						E5	
SEV	Set Overflow bit in CCR						E7	
SMSK	Set Interupt Mask	E9*						
STA	Store AA in memory		77	78	79	7A		
STB	Store BA in memory		7B	7C	7D	7E		
STX	Store X in memory		7F	80	81	82		
SUBA	Subtract operand from AA	4D	4E	4F	50	51		
SUBB	Subtract operand from BA	52	53	54	55	56		
TAB	Transfer AA to BA						83	
TBA	Transfer BA to AA						84	
TBS	Transfer BA to SP						85	
TEST	Test ALU and RAM						EA	
TSB	Transfer SP to BA						86	
WAI	Wait for Interrupt						E1	

* Uses 1 Byte Immediate addressing (See Programmers Model of CCU)

INSTRUCTION SET

This section defines the mnemonics for each of the instructions as well as listing the opcodes for each of the valid addressing modes.

The following symbols are defined :

$	=	Hex
@	=	Octal
%	=	Binary
'	=	Single ASCII Character
None	=	Decimal
AA	=	A Accumulator
BA	=	B Accumulator
X	=	X Register
SP	=	Stack Pointer
PC	=	Program Counter
C	=	Carry bit in CCU
V	=	Overflow bit in CCU
N	=	Negative bit in CCU
Z	=	Zero bit in CCU
M	=	Memory Value (1 byte)
M:M+1	=	Memory Value (1 word)
op	=	Opcode (1 byte)
hh	=	High order byte
ll	=	Low order byte
co	=	2's complement offset
(+)	=	Logical Exclusive Or
•	=	Logical Exclusive Nor
°	=	Logical And
||	=	Logical Or
⇑⇓	=	Bit set or cleared during Instruction

Mnemonic : **ABA**
Definition : Add B accumulator to A accumulator
Function : Add B Accumulator to A Accumulator. Result is placed in A accumulator. B accumulator remains unchanged.

AA = AA + BA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($02)

Mnemonic : **ABX**
Definition : Add B accumulator to X register
Function : Add B Accumulator to X Index Register. Result is placed in X Index Register. B accumulator remains unchanged.

X = X + BA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($03)

Mnemonic : ADCA
Definition : Add operand and carry bit to A accumulator
Function : Add operand to A accumulator along with the C bit of the CCU and place the result in A accumulator.

AA = AA + M:M+1 + C

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

IMM	($0E)
DIR	($0F)
INDIR	($10)
EXT	($11)
INDX	($12)

Mnemonic : ADCB
Definition : Add operand and carry bit to B accumulator
Function : Add operand to B accumulator along with the C bit of the CCU and place the result in B accumulator.

BA = BA + M:M+1 + C

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

IMM	($13)
DIR	($14)
INDIR	($15)
EXT	($16)
INDX	($17)

Mnemonic : ADDA
Definition : Add operand to A accumulator
Function : Add operand to A accumulator and place the result in A accumulator.

AA = AA + M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

IMM	($04)
DIR	($05)
INDIR	($06)
EXT	($07)
INDX	($08)

Mnemonic : ADDB
Definition : Add operand to B accumulator
Function : Add operand to B accumulator and place the result in B accumulator.

BA = BA + M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

IMM	($09)
DIR	($0A)
INDIR	($0B)
EXT	($0C)
INDX	($0D)

Mnemonic : **ANDA**
Definition : Logicai AND operand with A accumulator
Function : Logical binary AND operand with A accumulator and place the resuit in A accumulator.

AA = AA ˙ M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

IMM	($87)
DIR	($88)
INDIR	($89)
EXT	($8A)
INDX	($8B)

Mnemonic : **ANDB**
Definition : Logical AND operand with B accumulator
Function : Logical binary AND operand with B accumulator and place the result in B accumulator.

BA = BA ˙ M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

IMM	($8C)
DIR	($8D)
INDIR	($8E)
EXT	($8F)
INDX	($90)

Mnemonic : **ASL** (Same as **LSL**)
Definition : Perform Arithmetic Shift Left on operand
Function : Shifts all bits of operand 1 place to the left. LSB is loaded with a zero. The C bit of the CCR contains the MSB.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

DIR	($A7)
INDIR	($A8)
EXT	($A9)
INDX	($AA)

Mnemonic : **ASLA** (Same as **LSLA**)
Definition : Perform Arithmetic Shift Left on A accumulator
Function : Shifts all bits of A accumulator 1 place to the left. LSB is loaded with a zero. The C bit of the CCR contains the MSB.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

INH	($AB)

Mnemonic : ASLB (Same as LSLB)
Definition : Perform Arithmetic Shift Left on B accumulator
Function : Shifts all bits of B accumulator 1 place to the left. LSB is loaded with a zero. The C bit of the CCR contains the MSB.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

INH	($AC)

Mnemonic : ASR
Definition : Perform Arithmetic Shift Right on operand
Function : Shifts all bits of operand 1 place to the right. MSB is held constant. The C bit of the CCR contains the LSB.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

DIR	($AD)
INDIR	($AE)
EXT	($AF)
INDX	($B0)

Mnemonic : **ASRA**
Definition : Perform Arithmetic Shift Right on A accumulator
Function : Shifts all bits of A accumulator 1 place to the right. The MSB is held constant. The C bit of the CCR contains the LSB.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($B1)

Mnemonic : **ASRB**
Definition : Perform Arithmetic Shift Right on B accumulator
Function : Shifts all bits of B accumulator 1 place to the right. The MSB is held constant. The C bit of the CCR contains the LSB.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($B2)

Mnemonic : BCC (Same as BHS)
Definition : Branch if Carry bit clear
Function : Tests the carry bit of the CCR and causes a branch if clear.

if C=0 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($CB)

Mnemonic : BCS (Same as BLO)
Definition : Branch if Carry bit set
Function : Tests the carry bit of the CCR and causes a branch if set.

if C=1 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($CC)

Mnemonic : BEQ
Definition : Branch if equal
Function : Tests the Zero bit of the CCR and causes a branch if set.

if Z=1 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($CD)

Mnemonic : BGE
Definition : Branch if greater than or equal (for signed numbers)
Function : Tests if last operation used a smaller operand in relation to a larger register value or if both operand and register were equal.

if N (+) V = 0	PC	=	PC + 2 + co
else	PC	=	PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($CE)

Mnemonic : BGT
Definition : Branch if greater than (for signed numbers)
Function : Tests if last operation used a smaller operand in relation to a larger register value.

if N (+) V ll Z= 0	PC	=	PC + 2 + co
else	PC	=	PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($CF)

Mnemonic : **BHI** (for unsigned numbers)
Definition : Branch if greater than (for unsigned numbers)
Function : Tests if last operation used a smaller operand in relation to a larger register value.

if C || Z = 0 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($D0)

Mnemonic : **BHS** (Same as BCC)
Definition : Branch if greater than or same (for unsigned numbers)
Function : Tests if last operation used a smaller operand in relation to a larger register value or if both operand and register were equal.

if C=0 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($CB)

Mnemonic : BITA
Definition : Test bits of A accumulator with operand
Function : Logical binary AND operand with A accumulator without saving the result. Only the CCR is modified.

AA ⁘ M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

IMM	($91)
DIR	($92)
INDIR	($93)
EXT	($94)
INDX	($95)

Mnemonic : BITB
Definition : Test bits of B accumulator with operand
Function : Logical binary AND operand with B accumulator without saving the result. Only the CCR is modified.

BA ⁘ M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

IMM	($96)
DIR	($97)
INDIR	($98)
EXT	($99)
INDX	($9A)

Mnemonic : BLE
Definition : Branch if less than or equal to (for signed numbers)
Function : Tests if last operation used a larger operand in relation to a smaller register value or if both the register value and the operand were equal.

if N (+) V ll Z = 1 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($D1)

Mnemonic : BLO (same as BCS)
Definition : Branch if less than (for unsigned numbers)
Function : Tests if last operation used a larger operand in relation to a smaller register value.

if C = 1 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($CC)

Mnemonic : BLS
Definition : Branch if less than or equal to (for unsigned numbers)
Function : Tests if last operation used a larger operand in relation to a smaller register value or if both the register and the operand were equal.

if C II Z = 1 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($D2)

Mnemonic : BLT
Definition : Branch if less than (for signed numbers)
Function : Tests if last operation used a larger operand in relation to a smaller register value.

if N (+) V = 1 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($D3)

Mnemonic : **BMI**
Definition : Branch if minus
Function : Tests the negative bit of the CCR and causes a branch if set.

if N = 1 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($D4)

Mnemonic : **BNE**
Definition : Branch if not equal
Function : Tests the zero bit of the CCR and causes a branch if clear.

if Z = 0 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($D5)

Mnemonic : BPL
Definition : Branch if positive
Function : Tests the negative bit of the CCR and causes a branch if clear.

if N = 0 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($D6)

Mnemonic : BRA
Definition : Branch Always
Function : Use offset to modify PC.

PC = PC + 2 + co

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($D7)

Mnemonic : BVC
Definition : Branch if Overflow bit clear
Function : Tests the overflow bit of the CCR and causes a branch if clear.

if V = 0 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($D8)

Mnemonic : BVS
Definition : Branch if Overflow bit set
Function : Tests the overflow bit of the CCR and causes a branch if set.

if V = 1 PC = PC + 2 + co
else PC = PC + 2

Condition Code Register :

N	Z	V	C
-	-	_	_

Valid address modes :
REL ($D9)

Mnemonic : CBA
Definition : Compare B accumulator to A accumulator
Function : Subtracts the B accumulator from the A accumulator but does not store the result. Neither A nor B accumulator is changed, but the condition codes are set.

AA - BA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($18)

Mnemonic : CLC
Definition : Clear Carry bit in CCR
Function : Clears the Carry bit in the CCR. All other values in the CCR are held constant.

C = 0

Condition Code Register :

N	Z	V	C
-	-	-	0

Valid address modes :
INH ($E2)

Mnemonic : CLR
Definition : Clear memory location (1 word)
Function : Writes a $0000 to the location specified by the user.

M:M+1 = $0000

Condition Code Register :

N	Z	V	C
0	1	0	0

Valid address modes :

DIR	($57)
INDIR	($58)
EXT	($59)
INDX	($5A)

Mnemonic : CLRA
Definition : Clear A accumulator
Function : Places $0000 in the A accumulator

AA = $0000

Condition Code Register :

N	Z	V	C
0	1	0	0

Valid address modes :

INH	($5B)

Mnemonic : CLRB
Definition : Clear B accumulator
Function : Places $0000 in the B accumulator

BA = $0000

Condition Code Register :

N	Z	V	C
0	1	0	0

Valid address modes :
INH ($5C)

Mnemonic : CLV
Definition : Clear Overflow bit in CCR
Function : Clears the Overflow bit in the CCR. All other values in the CCR are held constant.

V = 0

Condition Code Register :

N	Z	V	C
-	-	0	-

Valid address modes :
INH ($E4)

Mnemonic : **CMSK**
Definition : Clear Interrupt Mask
Function : Clears the Interrupt Mask. This will effectively prevent all interrupt requests from generating interrupts. To set any of the mask bits use the SMSK command.

MASK = 0

Condition Code Register :

N	Z	V	C
-	-	-	-

Valid address modes :
INH ($E8)

Mnemonic : **CMPA**
Definition : Compare operand to A accumulator.
Function : Subtracts the operand from the A accumulator but does not store the result. A accumulator is not changed, but the condition codes are set.

AA - M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
IMM ($19)
DIR ($1A)
INDIR ($1B)
EXT ($1C)
INDX ($1D)

Mnemonic : CMPB
Definition : Compare operand to B accumulator.
Function : Subtracts the operand from the B accumulator but does not store the result. B accumulator is not changed, but the condition codes are set.

BA - M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

IMM	($1E)
DIR	($1F)
INDIR	($20)
EXT	($21)
INDX	($22)

Mnemonic : COM
Definition : Perform 1's complement of operand (1 word)
Function : Perform 1's complement of the word at the location specified by the user. This is equivalent to inverting all of the bits.

M:M+1 = $FFFF - M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	1

Valid address modes :

DIR	($9B)
INDIR	($9C)
EXT	($9D)
INDX	($9E)

Mnemonic : **COMA**
Definition : Perform 1's complement of A accumulator
Function : Perform 1's complement of A accumulator. This is equivalent to inverting all of the bits.

AA = $FFFF - AA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	1

Valid address modes :
INH ($9F)

Mnemonic : **COMB**
Definition : Perform 1's complement of B accumulator
Function : Perform 1's complement of B accumulator. This is equivalent to inverting all of the bits.

BA = $FFFF - BA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	1

Valid address modes :
INH ($A0)

Mnemonic : CPX
Definition : Compare operand to X Index register
Function : Subtracts the operand from the X register but does not store the result. The X register is not changed, but the condition codes are set.

X - M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

IMM	($23)
DIR	($24)
INDIR	($25)
EXT	($26)
INDX	($27)

Mnemonic : DEC
Definition : Decrement an operand by 1
Function : Subtracts 1 from the word specified by the user.

(M:M+1) = (M:M+1) - 1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

DIR	($2B)
INDIR	($2C)
EXT	($2D)
INDX	($2E)

Mnemonic : DECA
Definition : Decrement A accumulator by 1
Function : Subtracts 1 from the A accumulator

AA = AA - 1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($28)

Mnemonic : DECB
Definition : Decrement B accumulator by 1
Function : Subtracts 1 from the B accumulator

BA = BA - 1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($29)

Mnemonic : DES
Definition : Decrement Stack Pointer by 1
Function : Subtracts 1 from Stack Pointer

SP = SP - 1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($2F)

Mnemonic : DEX
Definition : Decrement X register by 1
Function : Subtracts 1 from X register

X = X - 1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($30)

Mnemonic : DSI
Definition : Disable Interrupts.
Function : Disables interrupts to prevent peripherals from interrupting the execution of the current program.

Condition Code Register :

N	Z	V	C
-	-	-	-

Valid address modes :
INH ($E6)

Mnemonic : ENI
Definition : Enable interrupts.
Function : Enables interrupts to provide peripherals with the ability to interrupt the execution of the current program. Interrupt service routines must be written and their locations entered in the Interrupt vector table.

Condition Code Register :

N	Z	V	C
-	-	-	-

Valid address modes :
INH ($E3)

Mnemonic : EORA
Definition : Exclusive Or A accumulator and operand
Function : Perform logical Exclusive Or with A accumulator and Operand, storing the result in A accumulator.

AA = AA (+) M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

IMM	($A1)
DIR	($EB)
INDIR	($EC)
EXT	($ED)
INDX	($EE)

Mnemonic : EORB
Definition : Exclusive Or B accumulator and operand
Function : Perform logical Exclusive Or with B accumulator and Operand, storing the result in B accumulator.

BA = BA (+) M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	1

Valid address modes :

IMM	($A2)
DIR	($EF)
INDIR	($F0)
EXT	($F1)
INDX	($F2)

Mnemonic : IDIV
Definition : Divide A accumulator by B accumulator (signed operation)
Function : Perform 2's complement divide using A accumulator as the dividend and the B accumulator as the divisor. The quotient is placed in the A accumulator and the remainder is placed in the B accumulator. Division by zero produces $FFFF in the Quotient and sets the overflow flag. The remainder is indeterminate. Also, if the most negative number (-32768) is divided by (-1) the overflow bit will be set and (-32768) will remain in AA.

AA ÷ BA; AA = Quotient; BA = Remainder

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

INH	($A5)

Mnemonic : **INC**
Definition : Increment an operand by 1
Function : Adds 1 to the word specified by the user.

(M:M+1) = (M:M+1) + 1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

DIR	($34)
INDIR	($35)
EXT	($36)
INDX	($37)

Mnemonic : **INCA**
Definition : Increment A accumulator by 1
Function : Adds 1 to the A accumulator

AA = AA + 1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

INH	($31)

Mnemonic : **INCB**
Definition : Increment B accumulator by 1
Function : Adds 1 to the B accumulator

BA = BA + 1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($32)

Mnemonic : **INS**
Definition : Increment Stack Pointer by 1
Function : Adds 1 to the Stack Pointer

SP = SP + 1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($38)

Mnemonic : **INX**
Definition : Increment X Register by 1
Function : Adds 1 to the X register.

$X = X + 1$

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($39)

Mnemonic : JMP
Definition : Jump to new location and execute program.
Function : Loads PC with operand. Previous PC value is lost

PC = M:M+1

Condition Code Register :

N	Z	V	C
-	-	-	-

Valid address modes :
INDIR ($DB)
EXT ($DC)

Mnemonic : JSR
Definition : Jump to subroutine and execute program.
Function : Loads PC with operand. Previous PC value is placed in memory at location designated by the Stack Pointer. The SP is then decremented by 2 This operation is matched with the RTS command which replaces PC with original value. Note that registers will not be saved during subroutine operations.

STACK = PC ; SP = SP - 2;
PC = M:M+1

Condition Code Register :

N	Z	V	C
-	-	-	-

Valid address modes :
INDIR ($DD)
EXT ($DE)

Mnemonic : LDA
Definition : Place operand in A accumulator
Function : Replace current contents of A accumulator with operand.
Original contents of A accumulator are lost
AA = M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

IMM	($5D)
DIR	($5E)
INDIR	($5F)
EXT	($60)
INDX	($61)

Mnemonic : LDB
Definition : Place operand in B accumulator
Function : Replace current contents of B accumulator with operand.
Original contents of B accumulator are lost
BA = M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

IMM	($62)
DIR	($63)
INDIR	($64)
EXT	($65)
INDX	($66)

Mnemonic : LDS
Definition : Place operand in Stack Pointer
Function : Replace current contents of Stack Pointer with operand.
Original contents of Stack pointer are lost
SP = M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

IMM	($67)
DIR	($68)
INDIR	($69)
EXT	($6A)
INDX	($6B)

Mnemonic : LDX
Definition : Place operand in X register
Function : Replace current contents of X register with operand.
Original contents of X register are lost

X = M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

IMM	($6C)
DIR	($6D)
INDIR	($6E)
EXT	($6F)
INDX	($70)

Mnemonic : **LSL** (Same as **ASL**)
Definition : Perform Logical Shift Left on operand
Function : Shifts all bits of operand 1 place to the left. LSB is loaded with a zero. The C bit of the CCR contains the MSB.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

DIR	($A7)
INDIR	($A8)
EXT	($A9)
INDX	($AA)

Mnemonic : **LSLA** (Same as **ASLA**)
Definition : Perform Logical Shift Left on A accumulator
Function : Shifts all bits of A accumulator 1 place to the left. LSB is loaded with a zero. The C bit of the CCR contains the MSB.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

INH	($AB)

Mnemonic : LSLB (Same as ASLB)
Definition : Perform Logical Shift Left on B accumulator
Function : Shifts all bits of B accumulator 1 place to the left. LSB is loaded with a zero. The C bit of the CCR contains the MSB.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($AC)

Mnemonic : LSR
Definition : Perform Logical Shift Right on operand
Function : Shifts all bits of operand 1 place to the right. MSB is replaced with zero. The C bit of the CCR contains the LSB.

Condition Code Register :

N	Z	V	C
0	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
DIR ($B9)
INDIR ($BA)
EXT ($BB)
INDX ($BC)

Mnemonic : LSRA
Definition : Perform Logical Shift Right on A accumulator
Function : Shifts all bits of A accumulator 1 place to the right. The MSB is replaced with zero. The C bit of the CCR contains the LSB.

Condition Code Register :

N	Z	V	C
0	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($BD)

Mnemonic : LSRB
Definition : Perform Logical Shift Right on B accumulator
Function : Shifts all bits of B accumulator 1 place to the right. The MSB is replaced with zero. The C bit of the CCR contains the LSB.

Condition Code Register :

N	Z	V	C
0	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($BE)

Mnemonic : **MUL**
Definition : Multiply two 16-bit, 2's complement numbers
Function : Multiply the 2's complement number contained in A accumulator with the 2's complement number contained in B accumulator. Result is a 32-bit number whose high order word is in A Accumulator and Low order word is in B accumulator.

AA:BA = AA * BA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($A6)

Mnemonic : **NEG**
Definition : Perform 2's complement of operand (1 word)
Function : Perform 2's complement of the word at the location specified by the user. Equivalent to 1's complement +1.

M:M+1 = $0000 - M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
DIR ($3D)
INDIR ($3E)
EXT ($3F)
INDX ($40)

Mnemonic : **NEGA**
Definition : Perform 2's complement of A accumulator
Function : Perform 2's complement of A accumulator. Equivalent to 1's complement +1.

AA = $0000 - AA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($3A)

Mnemonic : **NEGB**
Definition : Perform 2's complement of B accumulator
Function : Perform 2's complement of B accumulator. Equivalent to 1's complement +1.

BA = $0000 - BA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($3B)

Mnemonic : **NOP**
Definition : No Operation
Function : PC is incremented by 2 (as in normal instruction) otherwise all condition codes are held constant. This instruction is useful in debugging as well in implementing software delays.

Condition Code Register :

N	Z	V	C
-	-	-	-

Valid address modes :

INH	($01)

Mnemonic : **ORA**
Definition : Logically OR operand and A accumulator
Function : Logical binary OR operand with A accumulator and place the result in A accumulator.

AA = AA || M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

IMM	($B3)
DIR	($B4)
INDIR	($B5)
EXT	($B6)
INDX	($B7)

Mnemonic : ORB
Definition : Logically OR operand and B accumulator
Function : Logical binary OR operand with B accumulator and place the result in B accumulator.

BA = BA || M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

IMM	($B8)
DIR	($A3)
INDIR	($A4)
EXT	($DA)
INDX	($33)

Mnemonic : PSHA
Definition : Push A accumulator on stack.
Function : Place A accumulator contents in memory using Stack Pointer as address and decrement stack pointer by 2. The value of AA is unchanged.

STACK = AA ; SP = SP - 2;

Condition Code Register :

N	Z	V	C
-	-	-	-

Valid address modes :

INH	($71)

Mnemonic : PSHB
Definition : Push B accumulator on stack.
Function : Place B accumulator contents in memory using Stack Pointer as address and decrement stack pointer by 2. The value of BA is unchanged.

STACK = BA ; SP = SP - 2;

Condition Code Register :

N	Z	V	C
-	–	–	–

Valid address modes :
INH ($72)

Mnemonic : PSHX
Definition : Push X register on stack.
Function : Place X register contents in memory using Stack Pointer as address and decrement stack pointer by 2. The value of X is unchanged.

STACK = X ; SP = SP - 2;

Condition Code Register :

N	Z	V	C
-	–	–	–

Valid address modes :
INH ($73)

Mnemonic : **PULA**
Definition : Pull A accumulator from stack.
Function : Replace A accumulator contents with memory using Stack Pointer as address after incrementing stack pointer by 2. The original value of AA is lost.

SP = SP + 2; AA = STACK;

Condition Code Register :

N	Z	V	C
-	–	–	–

Valid address modes :
INH ($74)

Mnemonic : **PULB**
Definition : Pull B accumulator from stack.
Function : Replace B accumulator contents with memory using Stack Pointer as address after incrementing stack pointer by 2. The original value of BA is lost.

SP = SP + 2; BA = STACK;

Condition Code Register :

N	Z	V	C
-	–	–	–

Valid address modes :
INH ($75)

Mnemonic : PULX
Definition : Pull X accumulator from stack.
Function : Replace X accumulator contents with memory using Stack Pointer as address after incrementing stack pointer by 2. The original value of X is lost.

SP = SP + 2; X = STACK;

Condition Code Register :

N	Z	V	C
-	–	–	–

Valid address modes :

INH	($76)

Mnemonic : ROL
Definition : Rotate operand bits left through carry
Function : Shifts all bits of operand 1 place to the left. LSB is replaced with the carry bit. The C bit of the CCR contains the MSB at the end of the operation.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

DIR	($BF)
INDIR	($C0)
EXT	($C1)
INDX	($C2)

Mnemonic : ROLA
Definition : Rotate A accumulator bits left through carry
Function : Shifts all bits of A accumulator 1 place to the left. LSB is loaded from C bit of CCR. The C bit of the CCR contains the MSB at the end of the operation.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($C3)

Mnemonic : ROLB
Definition : Rotate B accumulator bit left through carry.
Function : Shifts all bits of B accumulator 1 place to the left. LSB is loaded from C bit of CCR. The C bit of the CCR contains the MSB at the end of the operation

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($C4)

Mnemonic : ROR
Definition : Rotate operand bits right through carry
Function : Shifts all bits of operand 1 place to the right. MSB is replaced with the carry bit. The C bit of the CCR contains the LSB at the end of the operation.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

DIR	($C5)
INDIR	($C6)
EXT	($C7)
INDX	($C8)

Mnemonic : RORA
Definition : Rotate A accumulator bits right through carry
Function : Shifts all bits of A accumulator 1 place to the right. MSB is loaded from C bit of CCR. The C bit of the CCR contains the LSB at the end of the operation.

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

INH	($C9)

Mnemonic : RORB
Definition : Rotate B accumulator bit right through carry.
Function : Shifts all bits of B accumulator 1 place to the right. MSB is loaded from C bit of CCR. The C bit of the CCR contains the LSB at the end of the operation

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($CA)

Mnemonic : RTI
Definition : Return from Interrupt Service Routine
Function : Replace all registers with original contents by pulling stored values from stack. Order of replacement is PC, X , B, A, CCR. The stack pointer will be reset to its original value in the process of removing items from the stack. Because the CCR is a 4 bit register, only the 4 LSB of the memory location are used. Current contents of each register is lost.

```
SP = SP + 2; PC  =    STACK;
SP = SP + 2; X   =    STACK;
SP = SP + 2; BA  =    STACK;
SP = SP + 2; AA  =    STACK;
SP = SP + 2; CCR =    STACK;
```

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($DF)

Mnemonic : RTS
Definition : Return from Subroutine
Function : Replace the PC with original contents by pulling stored value from stack. Current contents are lost.

SP = SP + 2; PC = STACK;

Condition Code Register :

N	Z	V	C
–	–	-	–

Valid address modes :
INH ($E0)

Mnemonic : SBA
Definition : Subtract B accumulator from A accumulator
Function : Subtracts the B accumulator from the A accumulator and stores the result in A accumulator. The contents of B accumulator are not modified.

AA = AA - BA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($42)

Mnemonic : **SBCA**
Definition : Subtract operand and carry bit from A accumulator
Function : Subtract operand from A accumulator along with the C bit of the CCU and place the result in A accumulator.

AA = AA - M:M+1 - C

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

IMM	($43)
DIR	($44)
INDIR	($45)
EXT	($46)
INDX	($47)

Mnemonic : **SBCB**
Definition : Subtract operand and carry bit from B accumulator
Function : Subtract operand from B accumulator along with the C bit of the CCU and place the result in B accumulator.

BA = BA - M:M+1 - C

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

IMM	($48)
DIR	($49)
INDIR	($4A)
EXT	($4B)
INDX	($4C)

Mnemonic : SBX
Definition : Subtract B accumulator from X register
Function : Subtract B Accumulator from X Index Register. Result is placed in X Index Register. B accumulator remains unchanged.

X = X - BA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($41)

Mnemonic : SEC
Definition : Set Carry bit in CCR
Function : Sets the Carry bit in the CCR. All other values in the CCR are held constant.

C = 1

Condition Code Register :

N	Z	V	C
-	-	-	1

Valid address modes :
INH ($E5)

Mnemonic : SEV
Definition : Set Overflow bit in CCR
Function : Sets the Overflow bit in the CCR. All other values in the CCR are held constant.

V = 1

Condition Code Register :

N	Z	V	C
-	-	1	-

Valid address modes :
INH ($E7)

Mnemonic : SMSK
Definition : Set Interrupt Mask
Function : Set Interrupt Mask to value specified in byte immediately following opcode. This is the only commands that uses the 1 byte Immediate addressing format. The MSB if the byte is corresponds to the highest priority interrupt. A 1 in a bit position enables the interrupt.

MASK = M

Condition Code Register :

N	Z	V	C
-	-	-	-

Valid address modes :
IMM (1 byte) ($E9)

Mnemonic : STA
Definition : Store A accumulator in Memory
Function : Place contents of A accumulator in memory at location specified by operand.

M:M+1 = AA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

DIR	($77)
INDIR	($78)
EXT	($79)
INDX	($7A)

Mnemonic : STB
Definition : Store B accumulator in Memory
Function : Place contents of B accumulator in memory at location specified by operand.

M:M+1 = BA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

DIR	($7B)
INDIR	($7C)
EXT	($7D)
INDX	($7E)

Mnemonic : STX
Definition : Store X register in Memory
Function : Place contents of X register in memory at location specified by operand.

M:M+1 = X

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

DIR	($7F)
INDIR	($80)
EXT	($81)
INDX	($82)

Mnemonic : SUBA
Definition : Subtract operand from A accumulator
Function : Subtract operand from A accumulator and place the result in A accumulator.

AA = AA - M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

IMM	($4D)
DIR	($4E)
INDIR	($4F)
EXT	($50)
INDX	($51)

Mnemonic : SUBB
Definition : Subtract operand from B accumulator
Function : Subtract operand from B accumulator and place the result in B accumulator.

BA = BA - M:M+1

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :

IMM	($52)
DIR	($53)
INDIR	($54)
EXT	($55)
INDX	($56)

Mnemonic : TAB
Definition : Transfer A accumulator to B accumulator.
Function : Transfer contents of A accumulator to B accumulator. The original contents of B accumulator are lost. The contents of A accumulator are not changed.

BA = AA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :

INH	($83)

Mnemonic : TBA
Definition : Transfer B accumulator to A accumulator.
Function : Transfer contents of B accumulator to A accumulator. The original contents of A accumulator are lost. The contents of B accumulator are not changed.

AA = BA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :
INH ($84)

Mnemonic : TBS
Definition : Transfer B accumulator to Stack Pointer
Function : Transfer contents of B accumulator to Stack Pointer. The original contents of the Stack Pointer are lost. The contents of B accumulator are not changed.

SP = BA

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :
INH ($85)

Mnemonic : TEST

Definition : Diagnostics test of ALU and RAM

Function : Series of operations are performed in the ALU to determine if it is functioning properly. If successful, the computer will then begin writing and reading known RAM locations. If this is successful, the program execution will continue. Otherwise, the location of the bad RAM will be continuously forced onto the data bus. To determine this value, use an LED arrangement as described in the Diagnostics section.

Condition Code Register :

N	Z	V	C
-	-	-	-

Valid address modes :

INH	($EA)

Mnemonic : TSB
Definition : Transfer Stack Pointer to B accumulator.
Function : Transfer contents of Stack Pointer to B accumulator. The original contents of B accumulator are lost. The contents of the Stack Pointer are not changed.

BA = SP

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	0	⇑⇓

Valid address modes :
INH ($86)

Mnemonic : **WAI**
Definition : Wait for Interrupt
Function : Place all registers onto stack and wait for interrupt. Registers are pushed onto stack in this order : CCR, AA, BA, X, PC

STACK = CCR; SP = SP - 2;
STACK = AA; SP = SP - 2;
STACK = BA; SP = SP - 2;
STACK = X; SP = SP - 2;
STACK = PC; SP = SP - 2;

Condition Code Register :

N	Z	V	C
⇑⇓	⇑⇓	⇑⇓	⇑⇓

Valid address modes :
INH ($E1)

MICROCODE - FORMAT AND DEFINITIONS

The following terms are defined as a set of mnemonics for the actions being performed by the data in the pipeline register :

Clear μ-status register carry bit:

μc := 0;	CE_M^* := 1
	2904 I_{0-5} := 001100

Set μ-status register carry bit:

μc := 1;	CE_M^* := 1
	2904 I_{0-5} := 001101

Clear μ-status carry bit and update M-status register :

μc := 0; -> MR;	CE_M^* := 0
	2904 I_{0-5} := 001100

Set μ-status carry bit and update M-status register :

μc := 1; -> MR;	CE_M^* := 0
	2904 I_{0-5} := 001101

Update M-status register:

-> MR;	CE_M^* := 0
	2904 I_{0-5} := 111111

Goto new address:

goto ??;	29811 I_{0-4} := 1111
	A_0 - A_{11} := next address

Disable access to Negative bit in Machine Status Reg:

Mnd; 2904 E_N^* := 1

Disable access to Overflow bit in Machine Status Reg:

Mvd; 2904 E_V^* := 1

Disable access to Zero bit in Machine Status Reg:

Mzd; 2904 E_Z^* := 1

Disable access to Carry bit in Machine Status Reg:

Mcd; 2904 E_C^* := 1

Enable access to Negative bit in Machine Status Reg:

Mne; 2904 E_N^* := 0

Enable access to Overflow bit in Machine Status Reg:

Mve; 2904 E_V^* := 0

Enable access to Zero bit in Machine Status Reg:

Mze; 2904 E_Z^* := 0

Enable access to Carry bit in Machine Status Reg:

Mce; 2904 E_C^* := 0

Clear all enabled bits in Machine Status Register:

Clr MR; CE_M^* := 0
2904 I_{0-5} := 000011

Set all enabled bits in Machine Status Register:

Set MR;	CE_M^* :=	0	
	2904 I_{0-5} :=		000001

Set Interrupt mask equal to lower byte of Data Bus values:

RMSK;	2914 I_{0-3} :=		1110
	I_E^* :=	0	

Enable Interrupts:

irqe := 1;	2914 I_{0-3} :=		1111
	I_E^* :=	0	

Disable Interrupts:

irqe := 0;	2914 I_{0-3} :=		1101
	I_E^* :=	0	

Read value from external memory:

rd;	EME :=	1
	R/W^* :=	1
	RLE^* :=	0
	RE^* :=	1

Write value to external memory:

wr;	EME :=	1
	R/W^* :=	0
	RE^* :=	0
	RLE^* :=	1

Enable external memory:

en;	EME :=	1
	R/W^* :=	1
	RLE^* :=	0
	RE^* :=	1

Read internal memory:

intrd;	IME := 0 R/W* := 1 RLE* := 0 RE* := 1

Enable internal memory:

en;	IME := 0 R/W* := 1 RLE* := 0 RE* := 1

Enable command ROM and Jump to location specified;

cromen;	29811 I_{0-4} := 0010

Enable interupt ROM :

irqvecen;	$E_N := 1$

The following inputs determine the origin of the ALU operands :

R = MBR	$EA^* := 1$; $I_0 := 0$; $OE_B^* := 0$;
S = CCR	$EA^* := 0$; $I_0 := 0$; $OE_B^* := 1$;
R,S = RAM	$EA^* := 0$; $I_0 := 0$; $OE_B^* := 0$;

The following registers are referenced with the associated codes:

Register	Code for A_0 - A_3 (S), B_0 - B_3 (R):
a	0000
b	0001
c	0010
d	0011
e	0100
f	0101
g	0110
h (1)	0111
i (0)	1000
j (bmask)	1001
k (15)	1010
l (AA)	1011
m (BA)	1100
n (X)	1101
o (SP)	1110
p (PC)	1111

Remember that the register indexed by A_0 - A_3 (S), will be the register written to if the RE^* is signal is low when the 2903 clock is low.

The following functions are performed using the associated codes:

Function:	Code for 2903 I_1 -I_{5a} - I_{5b} - I_8 (Note: I_{5a} and I_{5b} will be identical except for sxtnd command)
S := S - R - 1 + μc;	111110001
S := R - S - 1 + μc;	111110010
S := R + S + μc;	111110011
S := S + μc;	111110100
S := R + μc;	111110110
S := ~S + μc;	111110101
S := ~R + μc;	111110111
S := band (R,S);	111111100
S := band (~R,S);	111111001
S := bor (R,S);	111111111
S := R (+) S;	111111011
Divide1(R,S)	101000000
Divide (R,S)	110000000
Divcor (R,S)	111000000
Mult (R,S)	001000000
Multlst (R,S)	011000000
sxtnd (S)	111010100

Function:	Code for 2904 I_6 - I_{10}
lshift(S)	10000
lshift(S,Q)	10100
rshift(S)	00000
arshift(S)	01110
rrtc(S)	01001
rltc(S)	11001

For the 2904 to provide the proper condition for the complex condition situations, the 2904 must be passed the following codes :

Function:	Code for 2904 I_0 - I_5
Mc	101010
~Mc	101011
Mz	100100
~Mz	100101
Mn (+) Mv	100010
Mn (+) Mv || Mz	100000
Mz || Mc	101000
~Mz && (Mv • Mn)	100001
~ Mz && ~ Mc	101001
Mn • Mv	100011
Mn	101110
~Mn	101111
Mv	100110
~Mv	100111
μz	010100
~μz	010101
μn	011110
~μn	011111

MICROCODE - ADDRESSING FORMAT

Since the first part of every instruction is related to the addressing mode it uses, a general discussion of addressing in microcode will be presented in this section. This will allow a greater emphasis on the microcode used to implement each instruction. A table of pipeline values that corresponds to the microcode format used to describe each command was presented before this section.

There are seven possible addressing modes for this instruction set. These modes are : Immediate, Direct, Indirect, Extended, Indexed, Inherent and Relative offset. Some modes are for specific types of instructions. For example, the Relative offset mode of addressing is specific for branch instructions, since it only passes a 1 byte signed offset that can be used to modify the PC if the necessary conditions are true.

Addressing modes that are used with operations that modify a location of memory, must retain the location of the operand to return the modified value to memory. This is done by placing the operand in scratch RAM register "a" and the address of the operand in scratch register "b" at the end of each addressing mode code. Each addressing mode is similar in this respect.

It must be noted that with the current memory decoding scheme, all memory access must be limited to an even address ($0000, $1212, $E000, etc...). If the read (or write) of an odd address is attempted, the data is shifted down one address. For example, if the word #$3050 was written to memory and the address for the write was $E345, the word would be passed to locations $E344:$E345. Location $E346 would not be modified. The programmer must take this into account while doing relative addressing as well, since the opcode always appears at an even address as well.

Each mode will be presented individually in the following paragraphs. In addition, the format used to replace a modified operand in memory will be discussed in the final paragraphs.

In this method of addressing a constant is passed to the ALU in the word directly after the microcode word. The format of this mode, as discussed in the Programmer's model, is : op 00 hh ll. The microcode for this addressing format is :

```
(1)  μc := 1;
(2)  PC := PC +1 + μc; μc := 0;
(3)  MAR := ~PC + μc; μc := 1; rd;
(4)  PC := PC + 1 + μc; μc := 0; rd;
(5)  a := ~MBR + μc; μc := 0;
```

Line (1) sets the carry bit for the next instruction. The 2903 allows for increment by 2, but the method for doing this is seen in line (2). This line increments the PC by 2, so the PC now holds the address of the operand (word). Line (2) also clears the carry bit for the next instruction. The next line inverts the PC and places it in the Memory Access Register. The inversion is necessary, because the 2917 bus interface will reinvert the address as it is placed on the bus. The inverted address would create problems in the I/O decode, so the CPU inverts the address before it is placed in the register.

Line (3) also engages the read line to enable the data to be piped onto the data bus. This signal must be held for two successive clock cycles to ensure that the data is valid. This is due to the choice of memory chips for both RAM and ROM. If better quality memory chips were used, this would not be necessary. However, since this project was under a time constraint, familiar parts were used rather than using a new, unfamiliar family of memory devices. This line also sets the μc bit for the next instruction.

Line (4) increments the PC by 2. This ensures that the PC will be pointing at the next opcode when it returns from this instruction. The final line loads the memory buffer register into scratch RAM register "a" in the 2903. The MBR value must be inverted because the 2917 inverts in either direction. Since the immediate

addressing mode has no specific location of operand, RAM register "b" is not used.

Note that 1 instruction (**SMSK**) uses a 1 byte immediate format in its implementation. The microcode for this addressing format is included in the microcode description and does not use the above microcode to retrieve the necessary operand.

Direct Addressing :

This addressing mode is used to access the first 256 bytes in the memory. Since the majority of this space is user available RAM, this addressing mode is ideal for I/O intensive application. The addressing format for this mode is : op ll. The microcode used for this addressing mode is :

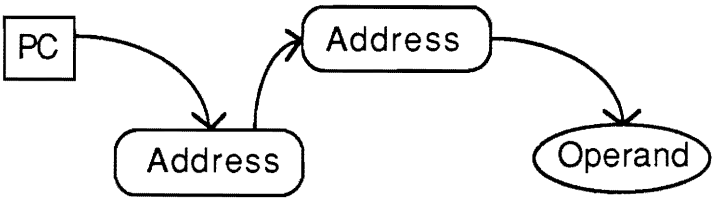
```
(1)    μc := 0; en;
(2)    b := band (~MBR,bmask); μc := 0;
(3)    MAR := ~b + μc; μc := 1; rd;
(4)    PC := PC + 1 + μc; μc := 0; rd;
(5)    a := ~MBR + μc; μc := 0;
```

Line (1) clears the carry for the first bit while maintaining the memory enable. This ensures that the data in MBR will still be the opcode word. The lower byte of this word is used to address the first 256 bytes of memory. Line (2) performs a binary AND of the word and the bmask. The bmask is a 1 word mask that contains 0's in the upper byte positions and 1's in the lower. This effectively creates an address of the opcode word that can address the necessary memory. Again, the MBR is inverted when passed into the ALU. The new address is inverted and placed in the MAR in line (3) and the read signal is engaged. Notice that the address of the operand is now in scratch register "b". This will ensure that the address will be available if the modified operand needs to be rewritten to this location.

Line (4) increments the PC as before to point at the next opcode word. The read signal remains engaged during this procedure. In line (5), the data is read from the MBR, inverted and placed in scratch register "a".

Indirect Addressing :

This addressing mode can be very useful if a full word pointer to memory is used to address the operand. Indirect addressing allows the programmer to use an address that points to an address where the operand is located (see figure below).



The format for this addressing mode is : op 00 hh ll. The microcode used to implement this mode is slightly longer than the rest of the addressing modes. This is due to the higher number of memory accesses needed to support this instruction. The microcode is shown below :

```
(1)     μc := 1;
(2)     PC := PC + 1 + μc;  μc := 0;
(3)     MAR := ~PC + μc; μc := 1; rd;
(4)     PC := PC + 1 + μc; μc := 0; rd;
(5)     MAR := MBR + μc; rd;
(6)     μc := 0; rd;
(7)     MAR := b := MBR + μc; μc := 0; rd;
(8)     b := ~b + μc; μc := 0; rd;
(9)     a := ~MBR + μc; μc := 0;
```

The first line of the code sets the carry bit for the next instruction. The PC must be incremented to address the operand. The PC is then inverted and placed in the MAR while the read signal is engaged.

While the CPU is waiting for the memory to become valid, the PC is again incremented to point at the next instruction. When the data is valid, it is passed directly to the MAR and the read signal is held active. The value does not need to be inverted since the 2917 will invert the address before it is placed on the bus.

The read signal is held until the data is valid, then the operand address is stored in "b" while the value is passed to the MAR. The read signal is still engaged during this process. During the delay for valid data, the value stored in register "b" is inverted and replaced. When the valid data is read from the MBR, the value is inverted and placed in register "a".

Extended Addressing:

Extended addressing allows the programmer to directly address the operand. The format for this instruction is : op 00 hh ll. This format is exactly like the indirect addressing format, except the address given is for the actual operand, not a pointer to the operand.

The microcode for this addressing mode is :

```
(1)     μc := 1;
(2)     PC := PC + 1 + μc; μc := 0;
(3)     MAR := ~PC + μc; μc := 1; rd;
(4)     PC := PC + 1 + μc; μc := 0; rd;
(5)     MBR := b := MBR + μc; μc := 0; rd
(6)     b := ~b + μc; μc := 0; rd;
(7)     a := ~MBR + μc; μc := 0;
```

This addressing mode is very similar to the indirect addressing mode, except 1 memory access is removed. As usual, the first instruction is to ensure the μc is in the proper state for the second instruction. Line (2) increments the PC to point at the address of the operand. The PC is then inverted and passed to the MAR, while

engaging the read signal. The read signal remains engaged as the PC is incremented again to point at the next opcode. When the data is valid, it is passed to the MAR and stored in "b" register without inverting it. During the next read cycle, the "b" register is inverted. The final instruction inverts the data from the MBR and stores the operand in register "a".

Indexed Addressing :

This addressing mode uses the index register "X" to provide a base address. A 1 byte offset (unsigned) is in the byte immediately following the opcode byte. This offset is temporarily added to the "X" register to provide the CPU with the address of the operand. The address mode format is : op ll. The microcode used to implement the instruction is :

```
(1)    μc := 0; en;
(2)    b := band (~MBR,bmask); μc :=0;
(3)    b := b + X + μc; μc := 0;
(4)    MAR := ~b + μc; μc := 1; rd;
(5)    PC := PC + 1 + μc; μc := 0; rd;
(6)    a := ~MBR + μc; μc := 0;
```

The first line sets the μc for the next instruction, while keeping the data on the data bus valid. The next instruction uses this data to determine the necessary offset by removing the upper byte. The lower byte is then added to the X register and the result is stored in the "b" register. The inverse of the "b" register is then stored in the MAR and the read signal is engaged. The read signal remains engaged as the PC is incremented to point at the next opcode. In the final cycle, the MBR is inverted and placed in register "a".

This mode is very similar to the direct addressing mode, except the offset is added to the X register to determine the address of the operand. In the direct addressing mode, the offset is effectively added to $0000.

Inherent Addressing :

from memory. Therefore, all instructions that are inherently addressed do not have any other instructions associated with there implementation, except those seen by their mnemonic in the microcode listing.

Relative Offset Addressing :

This mode is used with all branch instructions. The offset is found in the byte that follows the opcode. The byte is a signed offset that is added to the PC. This offset must have the sign bit extended through the upper byte to provide the ALU with a valid signed number. The format for this addressing mode is : op ll. The microcode for this addressing mode is :

(1) μc := 0; en;
(2) a := band (~MBR,bmask); μc := 0;
(3) a := sxtnd(a); μc := 0;

The first instruction maintains the data in the MBR while setting the μc for the next instruction. The next instruction masks out the upper byte of the instruction and replaces it with zeros. The last instruction, extends the sign bit through the upper byte in order to provide the ALU with a 16-bit signed representation of the 1 byte offset.

Returning the modified operand to memory :

Some instructions modify the operand and then replace it in memory. These instructions end with the expression "goto **". The location specified contains the following instructions which write the value contained in register "a" to the location found in register "b", then return to the beginning of the fetch routine :

```
(1)     MBR := ~a + μc; μc := 0;
(2)     MAR := ~b + μc; μc := 0; wr;
(3)     μc := 0; wr; goto 14;
```

MICROCODE - INSTRUCTIONS

The macro instruction set for this project has been modeled around the Motorola 6800 family of microprocessors. The largest differences have been in the interrupt structure (**SMSK** and **CMSK**). All commands that referenced Index Register Y were not implemented. There are a few new commands to manipulate the Stack Pointer and perform a RAM and ALU test.

The first part of the microprogram that merits discussion is the power-up routine. This routine is located at address zero in the ROM decode and is called by the 29811 JUMP TO ZERO instruction (see hardware POWER-UP). The microcode used at power-up is seen below (see Register Table for location of active registers and constants in 2903 RAM) :

```
(0)     μc := 1;
(1)     i := i - i - 1 + μc; μc := 0;
(2)     MAR := ~i + μc; μc := 0; intrd;
(3)     h := i + 1 + μc; μc := 0; intrd;
(4)     PC := ~MBR + μc; μc := 0;
(5)     MAR := ~h + μc; μc := 0; intrd;
(6)     a := h + 1 + μc; μc := 0; intrd;
(7)     SP := ~MBR + μc; μc := 0;
(8)     MAR := ~a + μc; μc := 0; intrd;
(9)     a := a + 1 + μc; μc := 0; intrd;
(10)    j := ~MBR + μc; μc := 0;
(11)    MAR := ~h + μc; μc := 0; intrd;
(12)    a := a + 1 + μc; μc := 0; intrd;
(13)    k := ~MBR + μc; μc := 0;
```

The first set of instructions loads a zero in register "i". This is then used to address the internal ROM (see Hardware - Power Up circuit) and read the first constant (default PC). During the read, a one is stored in register "h". This is used during the next read routine to

retrieve the next constant (SP). This continues until all constants are loaded into the 2903 RAM.

This should set up all necessary constants and load in default values for the PC ($C000) and the SP ($3FFE). The other values are constants necessary for start-up operation. The next code is the fetch, decode and execute section of the microcode. All instructions jump back to this location upon completion. Note that the first instruction checks for any active interrupt requests.

```
(14)    if nmi then goto 19;
(15)    if irq then goto 19; μc := 0;
(16)    MAR := ~PC + μc; rd;
(17)    μc := 0; rd;
(18)    cromen;
(19)    irqe := 0; μc := 0;
(20)    a := CCR; μc := 0;
(21)    MBR := ~a + μc;  μc := 0;
(22)    MAR := ~SP + μc; μc := 0; wr;
(23)    SP := SP - 1 - 1 + μc; μc := 0; wr;
(24)    MBR := ~AA + μc; μc := 0;
(25)    MAR := ~SP + μc; μc := 0; wr;
(26)    SP := SP - 1 - 1 + μc; μc := 0; wr;
(27)    MBR := ~BA + μc; μc := 0;
(28)    MAR := ~SP + μc; μc := 0; wr;
(29)    SP := SP - 1 - 1 + μc; μc := 0; wr;
(30)    MBR := ~X + μc; μc := 0;
(31)    MAR := ~SP + μc; μc := 0; wr;
(32)    SP := SP - 1 - 1 + μc; μc := 0; wr;
(33)    MBR := ~PC + μc; μc := 0;
(34)    MAR := ~SP + μc; μc := 0; wr;
(35)    SP := SP - 1 - 1 + μc; μc := 0; wr;
(36)    if nmi then goto 39; MAR := ~0 + μc; rd;
(37)    irqvecen; μc := 0;
(38)    MAR := MBR + μc; μc := 0; rd;
(39)    μc := 0; rd;
(40)    PC := ~MBR + μc; μc := 0; goto 14
```

This routine first checks for an active interrupt. If the interrupt is active, all interrupts are disabled. Next, all active registers are pushed onto the stack. Finally, the interrupt vector prom is enabled and the current interrupt vector is read into the "a" register. The interrupt service routine's location is then read from somewhere in the first 9 words of RAM. This location is loaded into the PC and the fetch routine is activated. Note that a programmer has the ability to

"nest" interrupts since the stack quite large and a programmer can reenable interrupts with the **ENI** command, but this is not recommended because the sequencing must be very exact.

The next pages contain the microcode used to implement the proposed macrocode.

Microcode listing of all instructions:

ABA	μc := 0;
[INH (02)]	AA := AA + BA + μc; -> MR, μc := 1;
	PC := PC + 1 + μc; μc := 0; goto 14;

Carry bit is cleared. AA is replaced with AA + BA + μc. Machine Register is updated. PC is incremented. Jump to Fetch.

ABX	μc := 0;
[INH (03)]	X := BA + X + μc; -> MR; μc := 1;
	PC := PC + 1 + μc; μc := 0; goto 14;

ADCA	MR -> μR;
[IMM (0E)]	AA := AA + a + μc; -> MR; μc := 0; goto 14;
[DIR (0F)]	
[INDIR (10)]	
[EXT (11)]	
[INDX (12)]	

Same as most addition. However, the Machine register's contents replace the current contents of the μ register. This provides the correct carry bit.

ADCB	MR -> μR;
[IMM (13)]	BA := BA + a + μc; -> MR; μc := 0; goto 14;
[DIR (14)]	
[INDIR (15)]	
[EXT (16)]	
[INDX (17)]	

ADDA	AA := AA + a + μc; -> MR; μc := 0; goto 14;
[IMM (04)]	
[DIR (05)]	
[INDIR (06)]	
[EXT (07)]	
[INDX (08)]	

ADDB	BA := BA + a + μc; -> MR; μc := 0; goto 14;
[IMM (09)]	
[DIR (0A)]	
[INDIR (0B)]	
[EXT (0C)]	
[INDX (0D)]	

ANDA	AA := band(AA,a); -> MR; μc := 0; goto 14;

[IMM (87)]
[DIR (88)]
[INDIR (89)]
[EXT (8A)]
[INDX (8B)]

ANDB — BA := band(BA,a); -> MR; µc := 0; goto 14;
[IMM (8C)]
[DIR (8D)]
[INDIR (8E)]
[EXT (8F)]
[INDX (90)]

ASL/LSL — µc := 0;
[DIR (A7)] — a := lshift (a); -> MR; µc := 0; goto **;
[INDIR (A8)]
[EXT (A9)]
[INDX (AA)]

Left shift operand replacing LSB with 0.

ASLA/LSLA — µc := 0;
[INH (AB)] — AA := lshift (AA); -> MR; µc := 1;
PC := PC + 1 + µc; µc := 0; goto 14;

ASLB/LSLB — µc := 0;
[INH (AC)] — BA := lshift (BA); -> MR; µc := 1;
PC := PC + 1 + µc; µc := 0; goto 14;

ASR — µc := 0;
[DIR (AD)] — a := arshift (a); -> MR; µc := 0; goto 14;
[INDIR (AE)]
[EXT (AF)]
[INDX (B0)]

Right shift operand while maintaining sign.

ASRA — µc := 0;
[INH (B1)] — AA := arshift (AA); -> MR; µc := 1;
PC := PC + 1 + µc; µc := 0; goto 14;

ASRB — µc := 0;
[INH (B2)] — BA := arshift (BA); -> MR; µc := 1;
PC := PC + 1 + µc; µc := 0; goto 14;

BCC/BHS — uc := 1; if Mc then goto (1);

[REL (CB)]

PC := PC + a + μc; μc =0;
(1) PC := PC + 1 + μc; μc =0; goto 14;

If test is true, then simply increment PC and move to the next instruction. Otherwise, Add offset to PC and continue.

BCS/BLO
[REL (CC)]

μc := 1; if ~Mc then goto (1);
PC := PC + a + μc; μc =0;
(1) PC := PC + 1 + μc; μc =0; goto 14;

BEQ
[REL (CD)]

μc := 1; if ~Mz then goto (1);
PC := PC + a + μc; μc :=0;
(1) PC := PC + 1 + μc; goto 14;

BGE
[REL (CE)]

μc := 1; if (Mn (+) Mv) then goto (1);
PC := PC + a + μc; μc =0;
(1) PC := PC + 1 + μc; μc := 0; goto 14;

BGT
[REL (CF)]

μc := 1; if ((Mn (+) Mv) + Mz) then goto (1);
PC := PC + a + μc; μc =0;
(1) PC := PC + 1 + μc; μc := 0; goto 14;

BHI
[REL (D0)]

μc := 1; if (Mz || Mc) then goto (1);
PC := PC + a + μc; μc =0;
(1) PC := PC + 1 + μc; μc := 0; goto 14

BITA
[IMM (91)]
[DIR (92)]
[INDIR (93)]
[EXT (94)]
[INDX (95)]

Y := band (AA,a); -> MR; uc := 0; goto 14;

Binary AND without storing the result. Y means that the RAM memory in the 2903 has not been enabled.

BITB
[IMM (96)]
[DIR (97)]
[INDIR (98)]
[EXT (99)]
[INDX (9A)]

Y := band (BA,a); -> MR; μc := 0; goto 14;

BLE
[REL (D1)]

μc := 1; if (~Mz && (Mv • Mn)) then goto (1);
PC := PC + a + μc; μc =0;
(1) PC := PC + 1 + μc; μc := 0; goto 14;

```
BLS                          μc := 1; if (~Mz && ~Mc) then goto (1);
[REL (D2)]                   PC := PC + a + μc; μc =0;
                         (1) PC := PC + 1 + μc; μc := 0; goto 14;

BLT                          μc := 1; if (Mn • Mv) then goto (1);
[REL (D3)]                   PC := PC + a + μc; μc =0;
                         (1) PC := PC + 1 + μc; μc := 0; goto 14;

BMI                          μc := 1; if (~Mn) then goto (1);
[REL (D4)]                   PC := PC + a + μc; μc =0;
                         (1) PC := PC + 1 + μc; μc := 0; goto 14;

BNE                          μc := 1; if (Mz) then goto (1);
[REL (D5)]                   PC := PC + a + μc; μc =0;
                         (1) PC := PC + 1 + μc; μc := 0; goto 14;

BPL                          μc := 1; if (Mn) then goto (1);
[REL (D6)]                   PC := PC + a + μc; μc =0;
                         (1) PC := PC + 1 + μc; μc := 0; goto 14;

BRA                          μc := 1;
[REL (D7)]                   PC := PC + a + μc; μc =0;
                             PC := PC + 1 + μc; μc := 0; goto 14;

BVC                          μc := 1; if (Mv) then goto (1);
[REL (D8)]                   PC := PC + a + μc; μc =0;
                         (1) PC := PC + 1 + μc; μc := 0; goto 14;

BVS                          μc := 1; if (~Mv) then goto (1);
[REL (D9)]                   PC := PC + a + μc; μc =0;
                         (1) PC := PC + 2 + μc; μc := 0; goto 14;

CBA                          μc := 1;
[INH (18)]                   PC := PC + 1 + μc; μc := 1;
                             c := AA - BA -1 + μc; -> MR; μc := 0; goto 14;

CLC                          μc := 1; Mnd; Mvd; Mzd;
[INH (E2)]                   PC := PC + 1 + μc; Clr MR;
                             μc := 0; Mne; Mve; Mze; goto 14;
```

Clear the machine register's carry bit by disabling everything but the carry and clearing the machine register.

```
CLR                          μc := 0;
[DIR (57)]                   a := 0 + μc; -> MR; μc := 0; goto **;
```

[INDIR (58)]
[EXT (59)]
[INDX (5A)]

CLRA
[INH (5B)]

μc := 1;
PC := PC + 1 + μc; μc := 0;
AA := 0 + μc; -> MR; μc := 0; goto 14;

CLRB
[INH (5C)]

μc := 1;
PC := PC + 1 + μc; μc := 0;
BA := 0 + μc; -> MR; μc := 0; goto 14;

CLV
[INH (E4)]

μc := 1; Mnd; Mcd; Mzd;
PC := PC + 1 + μc; Clr MR;
μc := 0; Mne; Mce; Mze; goto 14;

CMSK
[INH (E8)]

μc := 1;
PC := PC + 1 + μc; μc := 0;
MBR := ~0 + μc; μc := 0;
RMSK; μc := 0; goto 14;

Set the Interrupt mask by placing the data on the data bus, then signalling the 2914 to read it. In this case, the data is zero. No interupts enabled.

CMPA
[IMM (19)]
[DIR (1A)]
[INDIR (1B)]
[EXT (1C)]
[INDX (1D)]

μc := 1;
Y := AA - a - 1 + μc; -> MR; μc := 0; goto 14;

Subtract the operand from the accumulator and do not store the result.

CMPB
[IMM (1E)]
[DIR (1F)]
[INDIR (20)]
[EXT (21)]
[INDX (22)]

μc := 1;
Y := BA - a - 1 + μc; -> MR; μc := 0; goto 14;

COM
[DIR (9B)]
[INDIR (9C)]
[EXT (9D)]
[INDX (9E)]

μc := 0;
a := ~a + μc; -> MR; μc := 0; goto 14;

Perform the 1's complement of the operand and replace it with the new value.

COMA [INH (9F)]	µc := 1; PC := PC + 1 + µc; µc := 0; AA := ~AA + µc; -> MR; µc := 0; goto 14;
COMB [INH (A0)]	µc := 1; PC := PC + 1 + µc; µc := 0; BA := ~a + µc; -> MR; µc := 0; goto 14;
CPX [IMM (23)] [DIR (24)] [INDIR (25)] [EXT (26)] [INDX (27)]	µc := 1; c := X - a - 1 + µc; -> MR; µc := 0; goto 14;
DEC [DIR (2B)] [INDIR (2C)] [EXT (2D)] [INDX (2E)]	a := a - 0 - 1 + µc; -> MR; µc := 0; goto 14;
DECA [INH (28)]	µc := 1; PC := PC + 1 + µc; µc := 0; AA := AA - 0 - 1 + µc; -> MR; µc := 0; goto 14;
DECB [INH (29)]	µc := 1; PC := PC + 1 + µc; µc := 0; BA := BA - 0 - 1 + µc; -> MR; µc := 0; goto 14;
DECS [INH (2F)]	µc := 1; PC := PC + 1 + µc; µc := 0; SP := SP - 0 - 1 + µc; -> MR; µc := 0; goto 14;
DECX [INH (30)]	µc := 1; PC := PC + 1 + µc; µc := 0; X := X - 0 - 1 + µc; -> MR; µc := 0; goto 14;
DSI [INH (E6)]	µc := 1; IRQE := 0; PC := PC + 1 + µc; µc := 0; goto 14;

Disable the interrupts through a command to the 2914.

ENI	µc := 1; IRQE := 1;

```
[INH (E3)]                      PC := PC + 1 + μc; μc := 0; goto 14;
```

Enable the interrupts through a command to the 2914.

```
EORA                            μc := 1;
[INH (A1)]                      PC := PC + 1 + μc; μc := 0;
                                AA := AA (+) a; -> MR; μc := 0; goto 14;

EORB                            μc := 1;
[INH (A2)]                      PC := PC + 1 + μc; μc := 0;
                                BA := BA (+) a; -> MR; μc := 0; goto 14;

IDIV                            μc := 0;
[INH (A5)]                      b := 15 - 1 - 1 + μc; μc := 0;
                                a := BA + μc; μc := 0;
                                if μz then (9); Q := AA + μc; μc := 0;
                                if ~μn then goto (1); a := 0 + μc; μc := 0;
                                a := a - 0 - 1 + μc; μc := 0;
                             (1) Q := lshift(Q); a := lshift (a);
                                Divide1 (a,BA);
                             (2) Divide (a,BA); μc := 0; -> MR;
                                b := b - 0 - 1 + μc;
                                if ~μz then goto (2); MR -> μR;
                                DivCor (a,BA); μc := 0; -> MR;
                                Y := a + μc;
                                if μz then goto (3); Y := band (a,AA); μc := 1;
                                if ~μn then goto (4); Y:= BA - a - 1 + μc; μc := 0;
                                if μz then goto (5); Y := BA + a + μc; μc := 0;
                                if μz then goto (6); Y := Q + μc; μc := 1;
                             (3) AA := Q - 0 - 1 + μc; goto (9); -> MR;
                             (4) Y := bor (a,AA); μc := 1;
                                if ~μn then goto (3); b := Q - 0 - 1 + μc;
                                if μn := then goto (8); μc := 0;
                             (7) a := a + BA + μc; μc := 0;
                                Q := Q - 0 - 1 + μc; μc := 1; goto (3)
                             (5) b := Q + μc;
                             (6) if μn then goto (7); μc := 0;
                             (8) a := a - BA - 1 + μc; μc := 0;
                                a := a + 1 + μc; μc := 0;
                                Q := Q + 1 + μc; goto (3);
                             (9) μc := 1;
                                PC := PC + 1 + μc; μc := 0; goto 14;
```

Perfom the division by using the capabilities of the 2903 instructions. If the division is not even, i.e. the remainder is not zero, the remainder and the quotient must be "corrected". The rest of the routine makes sure that the sign of the quotient and remainder are correct.

INC [DIR (34)] [INDIR (35)] [EXT (36)] [INDX (37)]	a := a + 1 + µc; -> MR; µc := 0; **;
INCA [INH (31)]	µc := 0; AA := AA + 1 + µc; -> MR; µc := 1; PC := PC + 1 + µc; µc := 0; goto 14;
INCB [INH (32)]	µc := 0; BA := BA +1 + µc; -> MR; µc := 1; PC := PC + 1 + µc; µc := 0; goto 14;
INCS [INH (38)]	µc := 0; SP := SP + 1 + µc; -> MR; µc := 1; PC := PC + 1 + µc; µc := 0; goto 14;
INCX [INH (39)]	µc := 0; X := X + 1 + µc; -> MR; µc := 1; PC := PC + 1 + µc; µc := 0; goto 14;
JMP [INDIR (DB)] [EXT (DC)]	PC := a; goto 14;
JSR [INDIR (DD)] [EXT (DE)]	MBR := ~PC + µc; MAR := ~SP + µc; µc := 0; wr; SP := SP - 1 - 1 + µc; µc := 0; wr; PC := a + µc; µc := 0; goto 14;
LDA [IMM (5D)] [DIR (5E)] [INDIR (5F)] [EXT (60)] [INDX (61)]	AA := a + µc; µc := 0; -> MR; goto 14;
LDB [IMM (62)] [DIR (63)] [INDIR (64)] [EXT (65)] [INDX (66)]	BA := a + µc; µc := 0; -> MR; goto 14;

```
LDS                                  SP := a + μc; μc := 0; -> MR; goto 14;
[IMM (67)]
[DIR (68)]
[INDIR (69)]
[EXT (6A)]
[INDX (6B)]

LDX                                  X := a + μc; μc := 0; -> MR; goto 14;
[IMM (6C)]
[DIR (6D)]
[INDIR (6E)]
[EXT (6F)]
[INDX (70)]

LSR                                  μc := 0;
[DIR (B9)]                           a := rshift (a); -> MR; μc := 0; goto 14;
[INDIR (BA)]
[EXT (BB)]
[INDX (BC)]

LSRA                                 μc := 0;
[INH (BD)]                           AA := rshift (AA); -> MR; μc := 1;
                                     PC := PC + 1 + μc; μc := 0; goto 14;

LSRB                                 μc := 0;
[INH (BE)]                           BA := rshift (BA); -> MR; μc := 1;
                                     PC := PC + 1 + μc; μc := 0; goto 14;

MUL                                  μc := 0;
[INH (A6)]                           a := 0 + μc; μc := 0;
                                     b := AA + μc; μc := 0;
                                     c := BA + μc; μc := 0;
                                     d := 15 + μc; μc := 0;
                                     Q := c + μc;
                                (1)  Mult (b,c); μc := 0; -> MR;
                                     d := d - 0 - 1 + μc;
                                     if μz then goto (1); MR -> μR;
                                     MultLst (b,c); μc := 0; -> MR;
                                     AA := a + μc; μc := 0;
                                     BA := Q + μc; μc := 1;
                                     PC := PC + 1 + μc; μc := 0; goto 14;
```

The multiply instruction is very easy since the 2903 does all of the mathemetics. The routi merely places the operands into the first scratch registers and performs the 2's complemer multiply routine on them 15 times. The 2's complement last cycle correction is necessary t ensure that the sign is correct.

```
NEG                          µc := 1;
[DIR (3D)]                   a := 0 - a - 1 + µc; -> MR; µc := 0; goto 14
[INDIR (3E)]
[EXT (3F)]
[INDX (40)]

NEGA                         µc := 1;
[INH (3A)]                   AA := 0 - AA - 1 + µc; -> MR; µc := 1;
                             PC := PC + 1 + µc; µc := 0; goto 14;

NEGB                         µc := 1;
[INH (3B)]                   BA := 0 - BA - 1 + µc; -> MR; µc := 1;
                             PC := PC + 1 + µc; µc := 0; goto 14;

NOP                          µc := 1;
[INH (01)]                   PC := PC + 1 + µc; µc := 0; goto 14;

ORA                          µc := 1;
[INH (A3)]                   PC := PC + 1 + µc; µc := 0;
                             AA := bor (AA,a); -> MR; µc := 0; goto 14;

ORB                          µc := 1;
[INH (A4)]                   PC := PC + 1 + µc; µc := 0;
                             BA := bor (BA,a); -> MR; µc := 0; goto 14;

PSHA                         µc := 1;
[INH (71)]                   PC := PC + 1 + µc; µc := 0;
                             MAR := ~SP + µc; µc := 0;
                             MBR := ~AA + µc; µc := 0; wr;
                             SP := SP - 1 - 1 + µc; µc := 0; wr; goto 14;

PSHB                         µc := 1;
[INH (72)]                   PC := PC + 1 + µc; µc := 0;
                             MAR := ~SP + µc; µc := 0;
                             MBR := ~BA + µc; µc := 0; wr;
                             SP := SP - 1 - 1 + µc; µc := 0; wr; goto 14;

PSHX                         µc := 1;
[INH (73)]                   PC := PC + 1 + µc; µc := 0;
                             MAR := ~SP + µc; µc := 0;
                             MBR := ~X + µc; µc := 0; wr;
                             SP := SP - 1 - 1 + µc; µc := 0; wr; goto 14;

PULA                         µc := 1;
[INH (74)]                   SP := SP + 1 + µc;
```

```
                                MAR := ~SP + µc; µc := 1; rd;
                                PC := PC + 1 + µc; µc := 0; rd;
                                AA := ~MBR + µc; µc := 0; goto 14;

PULB                            µc := 1;
[INH (75)]                      SP := SP + 1 + µc;
                                MAR := ~SP + µc; µc := 1; rd;
                                PC := PC + 1 + µc; µc := 0; rd;
                                BA := ~MBR + µc; µc := 0; goto 14;

PULX                            µc := 1;
[INH (76)]                      SP := SP + 1 + µc;
                                MAR := ~SP + µc; µc := 1; rd;
                                PC := PC + 1 + µc; µc := 0; rd;
                                X := ~MBR + µc; µc := 0; goto 14;

ROL                             MR -> µR;
[DIR (BF)]                      a := lrtc (a); -> MR; µc := 0; goto **
[INDIR (C0)]
[EXT (C1)]
[INDX (C2)]

ROLA                            MR -> µR;
[INH (C3)]                      AA := lrtc (AA); -> MR; µc := 1;
                                PC := PC + 1 + µc; µc := 0; goto 14;

ROLB                            MR -> µR;
[INH (C4)]                      BA := lrtc (BA); -> MR; µc := 1;
                                PC := PC + 1 + µc; µc := 0; goto 14;

ROR                             MR -> µR;
[DIR (C5)]                      a := rrtc (a); -> MR; µc := 0; goto **;
[INDIR (C6)]
[EXT (C7)]
[INDX (C8)]

ROLA                            MR -> µR;
[INH (C9)]                      AA := rrtc (AA); -> MR; µc := 1;
                                PC := PC + 1 + µc; µc := 0; goto 14;

ROLB                            MR -> µR;
[INH (CA)]                      BA := rrtc (BA); -> MR; µc := 1;
                                PC := PC + 1 + µc; µc := 0; goto 14;

RTI                             µc := 1;
[INH (DF)]                      SP := SP + 1 + µc; µc := 0;
```

```
                    MAR := ~SP + µc; µc := 1; rd;
                    SP := SP + 1 + µc; µc := 0; rd;
                    PC := ~MBR + µc; µc := 0;
                    MAR := ~SP + µc; µc := 1; rd;
                    SP := SP + 1 + µc; µc := 0; rd;
                    X := ~MBR + µc; µc := 0;
                    MAR := ~SP + µc; µc := 1; rd;
                    SP := SP + 1 + µc; µc := 0; rd;
                    BA := ~MBR + µc; µc := 0;
                    MAR := ~SP + µc; µc := 1; rd;
                    SP := SP + 1 + µc; µc := 0; rd;
                    AA := ~MBR + µc; µc := 0;
                    MAR := ~SP + µc; µc := 1; rd;
                    SP := SP + 1 + µc; µc := 0; rd;
                    a := ~MBR + µc; µc := 0;
                    CCR := a +µc; µc := 0; goto 14;
```

The return from interrupt replaces all active registers with values retrieved from the stack. This includes the CCR and the PC. Program execution resumes at the point it was interrupted.

```
RTS                 µc := 0;
[INH (E0)]          SP := SP + 1 + µc; µc := 0;
                    MAR := SP + µc; µc := 0; rd;
                    SP := SP + 1 + µc; µc := 0; rd;
                    PC := MBR + µc; µc := 0; goto 14;

SBA                 µc := 1;
[INH (42)]          PC := PC + 1 + µc; µc := 1;
                    AA := AA - BA - 1 + µc; -> MR; µc := 0; goto 14;

SBCA                if Mc then goto (1); µc := 0;
[IMM (43)]          µc := 1;
[DIR (44)]        (1) AA := AA - a - 1 + µc; -> MR; µc := 0; goto 14;
[INDIR (45)]
[EXT (46)]
[INDX (47)]

SBCB                if Mc then goto (1); µc := 0;
[IMM (48)]          µc := 1;
[DIR (49)]        (1) BA := BA - a - 1 + µc; -> MR; µc := 0; goto 14;
[INDIR (4A)]
[EXT (4B)]
[INDX (4C)]

SBX                 µc := 1;
```

[INH (41)]	PC := PC + 1 + µc; µc := 1; X := X - BA - 1 + µc; -> MR; µc := 0; goto 14;
SEC [INH (E5)]	µc := 1; Mnd; Mvd; Mzd; PC := PC + 1 + µc; Set MR; µc := 0; Mne; Mve; Mze; goto 14;
SEV [INH (E7)]	µc := 1; Mnd; Mcd; Mzd; PC := PC + 1 + µc; Set MR; µc := 0; Mne; Mce; Mze; goto 14;
SMSK [1 BYTE IMM (E9)]	µc := 0; en; MBR := band (MBR,bmask); µc := 1; RMSK; µc := 1; PC := PC + 1 + µc; µc := 0; goto 14;
STA [DIR (77)] [INDIR (78)] [EXT (79)] [INDX (7A)]	µc := 0; a := AA + µc; -> MR; µc := 0; goto 14;
STB [DIR (7B)] [INDIR (7C)] [EXT (7D)] [INDX (7E)]	µc := 0; a := BA + µc; -> MR; µc := 0; goto **;
STX [DIR (7F)] [INDIR (80)] [EXT (81)] [INDX (82)]	µc := 0; a := X + µc; -> MR; µc := 0; goto **
SUBA [IMM (4D)] [DIR (4E)] [INDIR (4F)] [EXT (50)] [INDX (51)]	µc := 1; AA := AA - a - 1 + µc; -> MR; µc := 0; goto 14;
SUBB [IMM (52)] [DIR (53)] [INDIR (54)] [EXT (55)]	µc := 1; BA := BA - a - 1 + µc; -> MR; µc := 0; goto 14;

[INDX (56)]

```
TAB                          μc := 1;
[INH (83)]                   PC := PC + 1 + μc; μc := 0;
                             BA := AA + μc; -> MR; μc := 0; goto 14;

TEST                         μc := 0;
[INH (EA)]                   a := ~0 + μc; μc := 0;
                             if ~μn then goto (1)
                         (2) a := lshift (a);
                             if μc then goto (2);
                             if ~μz then goto (1);
                             a := ~0 + μc; μc := 0;
                             if ~μn then goto (1)
                         (3) a := rshift (a);
                             if μc then goto (3);
                             if ~μz then goto (1); μc := 0;
                             a := 0 - 0 - 1 + μc;
                             if ~n then goto (1);
                             a := ~0 + μc;
                             a := rshift(a);
                             a := rshift(a); μc := 0;
                         (4) MAR := ~a + μc; rd;
                             rd; μc := 0;
                             b := ~MBR + μc; μc := 0; en;
                             MBR := 0 + μc; wr;
                             wr; μc := 0;
                             rd;
                             rd; μc := 0;
                             c := MBR + μc; μc := 0; en;
                         (5) if ~μz then goto (5); μc := 0; en;
                             MBR := ~b + μc; wr;
                             μc := 0; wr;
                             a := a - 0 - 1 + μc;
                             if μz then goto (4); μc := 1;
                             PC := PC + 1 + μc; μc := 0; goto 14
                             μc := 0;
                         (1) MBR := ~a + μc; en; goto (1)
```

The test routine performs several mathemetic operations, then checks the available RAM. This routine will take some time. Since every RAM location must be read and written twice

```
TBA                          μc := 1;
[INH (84)]                   PC := PC + 1 + μc; μc := 0;
                             AA := BA + μc; -> MR; μc := 0; goto 14;
```

```
TBS                          µc := 1;
[INH (85)]                   PC := PC + 1 + µc; µc := 0;
                             SP := BA + µc; -> MR; µc := 0; goto 14;

TSB                          µc := 1;
[INH (86)]                   PC := PC + 1 + µc; µc := 0;
                             BA := SP + µc; -> MR; µc := 0; goto 14;

WAI                          µc := 1;
[INH (E1)]                   PC := PC + 1 + µc; µc := 0;
                             a := CCR + µc; µc := 0;
                             MBR := ~a + µc; µc := 0;
                             MAR := ~SP + µc; µc := 0; wr;
                             SP := SP - 1 - 1 + µc; µc := 0; wr;
                             MBR := ~AA + µc; µc := 0;
                             MAR := ~SP + µc; µc := 0; wr;
                             SP := SP - 1 - 1 + µc; µc := 0; wr;
                             MBR := ~BA + µc; µc := 0;
                             MAR := ~SP + µc; µc := 0; wr;
                             SP := SP - 1 - 1 + µc; µc := 0; wr;
                             MBR := ~X + µc; µc := 0;
                             MAR := ~SP + µc; µc := 0; wr;
                             SP := SP - 1 - 1 + µc; µc := 0; wr;
                             MBR := ~PC + µc; µc := 0;
                             MAR := ~SP + µc; µc := 0; wr;
                             SP := SP - 1 - 1 + µc; µc := 0; wr;
                          (2) if irq then goto (1);
                             if nmi then goto (3); goto (2)
                          (3) MAR := ~0 + µc; rd; goto (4)
                          (1) irqe := 0;
                             irqvecen; µc := 0;
                             MAR := MBR + µc; µc := 0; rd;
                          (4) µc := 0; rd;
                             PC := ~MBR + µc; µc := 0; goto 14;
```

This routine places all of the active registers on the stack, just as if an interrupt had been acknowledged. The routine then waits for an interrupt. When an interrupt occurs, the routine disables interupts, loads the interrupt service routine location, and jumps to the fetch routine.

MICROCODE - SUMMARY

Multiplication and Division:

Two of the largest microprogram sections deal with multiplication and division. The 2903 is capable of performing both of these arithmetic functions, so this code takes advantage of the extra operands available. The code, while convoluted, does follow the necessary format to produce signed multiplication/division.

Addition and Subtraction with Overflow Halt:

To implement overflow halt, a programmer must test for the overflow bit being set after an arithmetic instruction. This can be done with the **BVS** command which can branch to an overflow service routine if the overflow is set.

Floating Point Addition and Subtraction:

Although floating point numbers are not supported in microcode, a macroprogrammer can easily implement addition and subtraction with a routine that would check for the smaller, signed exponent, then rotate the mantissa of that number right while adding 1 to the exponent until both exponents are equal, then adding or subtracting the mantissas. The direct memory access commands like INC and ASR would make implementation of such a macroprogram simple.

Diagnostics:

The macroinstruction **TEST** allows the user to test the ALU and RAM to see if all functions are working properly. If the ALU fails, the Data bus will hold the working contents of the ALU. If the RAM fails, the data bus will hold the location of the bad RAM. This data could easily be displayed to the user through the LED module discussed in the hardware specifications.

Conclusion

The design presented in this report meets all criteria designated in the project description. There were no real problems to report, and the following is a discussion of the details of the design.

Communications Bus:

There are three bus structures present in this design. A 16-bit data bus for data communication between the microprocessor and peripherals, a 16-bit address bus to specify memory/peripherals, and a 6-line control bus containing the IRQ, the NMI, two clock lines, and the external memory lines (enable and R/W*). Each bus has sufficient room for additional peripherals.

Microprocessor Module:

The microprocessor designed is a 16-bit machine (64K memory map) with a fairly broad instruction set. The microprocessor is capable of most arithmetic functions, including integer multiply and divide. In addition, with the available commands for memory shifts and rotates, a programmer could easily write code for floating point addition and subtraction. The processor uses available RAM for stack operations using an internal, user accessible, stack pointer.

RAM Module:

The RAM module specifically designed for this project uses a 6164 static RAM with decoding to address the first 8K of memory ($0000 - $3FFF). The RAM uses a buffer to interface with the data bus to provide sufficient drive. This same module design could be used to provide additional RAM at another location in memory.

EPROM Module:

The EPROM module specifically designed for this project uses a 2764 EPROM with decoding to address the last 8K of memory ($C000 - $FFFF). The EPROM uses a buffer to interface with the data bus to provide sufficient drive. This same module design could be used to provide addition EPROMs at another location in memory.

Note : All memory access requires that the programmer point to the even byte of an address to pull in a word of data, i.e. EVEN:ODD is valid, ODD:EVEN is not valid.

I/O Instructions:

Since all I/O is memory referenced, the decode process allows I/O devices to either transmit or receive data from the microprocessor. It must also be noted that since this memory referencing is inherent, the above requirement for memory access applies to an I/O module as well.

I/O Modules:

Several different I/O modules were designed, including ADC and DAC modules as well as drivers for LED's and inputs for switches. A UART interface was also designed to provide the microprocessor with a method of transmitting and receiving serial data. The SSR module will allow the microprocessor to interface with solid-state relays with optoisolators present to provide signal isolation. An additional input-output port was also added to allow for user-specific applications.

Software Design:

The instruction set chosen for this design is similar to the Motorola 6800 series of commands, with several additional commands to interface with the design specific devices present. The microprocessor instruction set supports integer mathematics, including multiplication and division. The floating point operations would be left up to the macroprogrammer, although the shift and rotate memory locations commands would make this task a simple one. The addition and subtraction with overflow halt could be implemented by branching to a service routine when the overflow flag is set after a mathematic instruction.

The diagnostic routine available to the user is a command that will allow the user to test the ALU and RAM memory that is known to be present in the memory map. The instruction is performed like any other macro instruction and does not affect condition flags. If a problem is present in the ALU, the instruction will place the

working data on the data bus and stop all processing. If a bad RAM location is found, the command will place the address of the bad location on the data bus, and stop all processing.

We feel confident that this design is viable, although more time would be beneficial in providing accurate calculations for timing and loading.

If this project were to be repeated, there would be several modifications. A separate bus would probably be added specifically for I/O and a bus transceiver used to interface the I/O bus with the data bus. A serial peripheral interface would also be added to enable the microprocessor to synchronously communicate with a set of peripherals.

In the software, several commands would probably be added to the interrupt structure to allow bit setting and clearing of the mask bits, as well as a way to read the current mask. The addressing decode scheme for macro commands would also be changed to allow the operand to be loaded by a generic addressing microroutine before passing the operand to a generic implementation routine. This would greatly reduce the amount of code in the command prom and would probably leave room for additional commands.

References

Advanced Micro-Devices. AMD Microprocessor and Microcontroller Databook. 1984.

Tanenbaum, Andrew S. Structured Computer Organization. Prentice Hall, 1990.

Sedra, Adel S., and Kenneth B. Smith. Microelectronic Circuits. 2nd ed. Chicago: Holt, Rinehart, and Winston, 1987.

Breeding, Kenneth J. Digital Design Fundamentals. Prentice Hall, 1989.

Scanlon, Leo J. IBM PC & XT Assembly Language. New York: Brady, 1985.

Datel. Data Acquisition and Conversion Handbook. 1979.

Motorola. M68HC11 Reference Manual. Rev 1. 1990.

Motorola. M68HC11EVB Evaluation Board User's Manual. 1986.

Motorola. MC68HC11F1 Technical Data Manual. Rev 2. 1990.

Motorola. Optoelectronics Device Data. Rev 3. 1989.

Motorola. Memory Device Data. Rev 7. 1991.

Texas Instruments. TTL Logic Data Book. 1988.

Microsoft. Microsoft Macro Assembler Manual. 1983.

Rector, Russell and George Alexy. The 8086 Book. McGraw-Hill, 1980.