AN ABSTRACT OF THE THESIS OF

Vicente Dy Loa (Name) for the Master of Science (Degree)

in Electrical and Electronics Engineering (Major) presented on July 31, 1972 (Date)

Title : LOGICAL DESIGN OF A SMALL DIGITAL COMPUTER USING ROM IN THE ARITHMETIC UNIT.

Abstract approved : Redacted for Privacy

Professor James H. Herzog

This thesis is concerned with the logic design of a small digital computer employing Read Only Memory(ROM) in a special type of arithmetic unit(AU). The AU is controlled by the ROM and is capable of performing a large number of operations. The performance of the unit with its associated ROM is explored.

In order to prove out the feasibility of utilizing such an arithmetic unit, a small computer is designed and examined. This computer can have a rich instruction set without complicated logic networks. Presented also is a brief discussion of related subjects such as microprogramming and a number of sample programs to illustrate the operation of the system.

Logical Design of a Small Digital Computer Using ROM in the Arithmetic Unit

by

Vicente Dy Loa

A THESIS

submitted to

Oregon State University

in partial fulfillment of

the requirements for the

degree of

Master of Science

June 1973

APPROVED:

Redacted for Privacy

Associate Professor of Electrical and Electronics Engineering

in charge of major

Redacted for Privacy

Head of Department of Electrical and Electronics Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented July 31, 1972

Typed by Vicente Dy Loa

ACKNOWLEDGEMENT

The author wishes to express special gratitude to Professor James H. Herzog for his initiation, guidance, and assistance which made this thesis possible.

TABLE OF CONTENTS

I. Introduction 1

II. Arithmetic Unit 7

III. Organization and Operation 17

Organization 17

Operation 19

IV. Logic Design 28

The Control Unit 28

I/O Data Transfer 32

The Memory Unit 36

R Register 37

Memory Address Register 38

Instruction Register 38

The ROM and RAM 40

Link 41

Accumulator 41

Program Counter 42

V. Conclusion 44

Bibliography 46

LOGICAL DESIGN OF A SMALL DIGITAL COMPUTER USING ROM IN THE ARITHMETIC UNIT

I. INTRODUCTION

The history of the calculating machine dates back a great many years. From primitive devices such as the abacus to modern electronic digital computers stretches a long interval in years and in technical accomplishment. As techniques and science progressed, ways of handling numbers and systems of operations were improved. Since the first generation of digital computers characterized by vacuum tubes to the present fourth generation characterized by large scale integration(LSI) technology, the design and construction of digital computers have gone through significant improvements. With the development of LSI technology, new ways of implementing the computer architecture employing functional building blocks have been evolved. To consider every aspect in this long process of evolution in computer technology is complex and beyond concern. Only subjects of related importance in connection with the proposed thesis will be briefly discussed.

A modern computer can be divided into five sections: the memory, control, arithmetic, input, and output. The control unit receives instructions from the memory and directs the operations of the entire machine. In the conventional approach, the control unit decodes the operation

field of each instruction and issues logic control signals to execute the instruction by specific gates throughout the system. The control unit is usually constructed by connecting a set of decoders and flip-flops in an untidy and unsystematic way. Generally, this type of control unit offers the most complexity but at low cost if the computer has a comprehensive instruction set. A decision to make the slightest modification to the instruction set could require a major modification of the whole structure. It is this complexity and lack of systematic structure in the design that prevents the designer or user from customizing the instruction set for his application.

Microprogramming was originally proposed as an alternative design procedure to the unsystematic way applied to conventional hardware. The concept was first proposed by M.V. Wilkes of Cambridge University[4]. The idea is to replace the hardware control by a stored logic control section using non-destructive Read Only Memory(ROM). Each word in the ROM represent a microinstruction. Each microinstruction controls signals according to the control code stored in that word. It will also select the address of the successor microinstruction during the operation. This feature eliminates all the fetch and decode cycles for every machine code in the microprogramed subroutine. Microprogramming is hardware implemented as software, its stored control has the ability

to alter the complete set of instruction without affecting the hardware design. By just changing the ROM microprogram storage, one can alter the instruction set. A conventional design would require a great modification to accomplish a similar instruction set modification.

A need for new minimization criteria in computer design has arisen due to LSI. LSI technology has made possible the fabrication of thousands of gates in a single chip. An efficient partitioning of the computer architecture into building blocks allows a minimum number of chip designs to be used as repetitiously as necessary with high degree of utilization of the circuitry on the chips.

The size and complexity of each chip is determined by the logic partitioning and limited by the number of pin connections. One candidate in the partitioning may include the universal logic block(ULB)[5]. ULB is a logic circuit with control inputs capable of providing Boolean functions of a fixed number of variables. The logic circuit consists of gates representing minterms of the variables. The realization of a Boolean function is performed by the selection of these minterms by varying the input terminal connections.

A ULB for a large number of variables such as six may be too complex and uneconomical to be built in a single chip because of the number of pin connections needed. A three variable ULB requires only 14 pin connections.

Usually when a machine has a comprehensive instruction set, the design of logic networks will also be very involved. The purpose of this thesis is to examine the characteristics of a small digital computer which can have many instructions and yet have simplicity in design by attempting to utilize ROM in a special type of arithmetic unit(AU). The AU will consist of identical cells where each cell is constructed of a particular type of ULB having 16 control inputs, three data inputs, and two data outputs. The ROM will be used as a function-select matrix to provide any Boolean function of three variables in each cell. This AU will be able to perform a large number of operations.

To test out the feasibility of using such an AU, a small computer will be designed with the proposed register organization shown in Figure 1.1. The computer's word length is chosen to be eight bits. Capacity of the memory unit will consist of 256 words which could be expanded if the word length is increased or a more elaborate addressing scheme is adopted. There are five basic registers in the computer, each of them will be eight bits. The Link is a special single bit register used to store overflow from the AU.

The ROM that is serving as a function-select matrix is specified to consist of 8 words by 16 bits, its contents are to be addressed by three bits from the Instruction Register. A RAM (Random Access Memory) with read/write capability

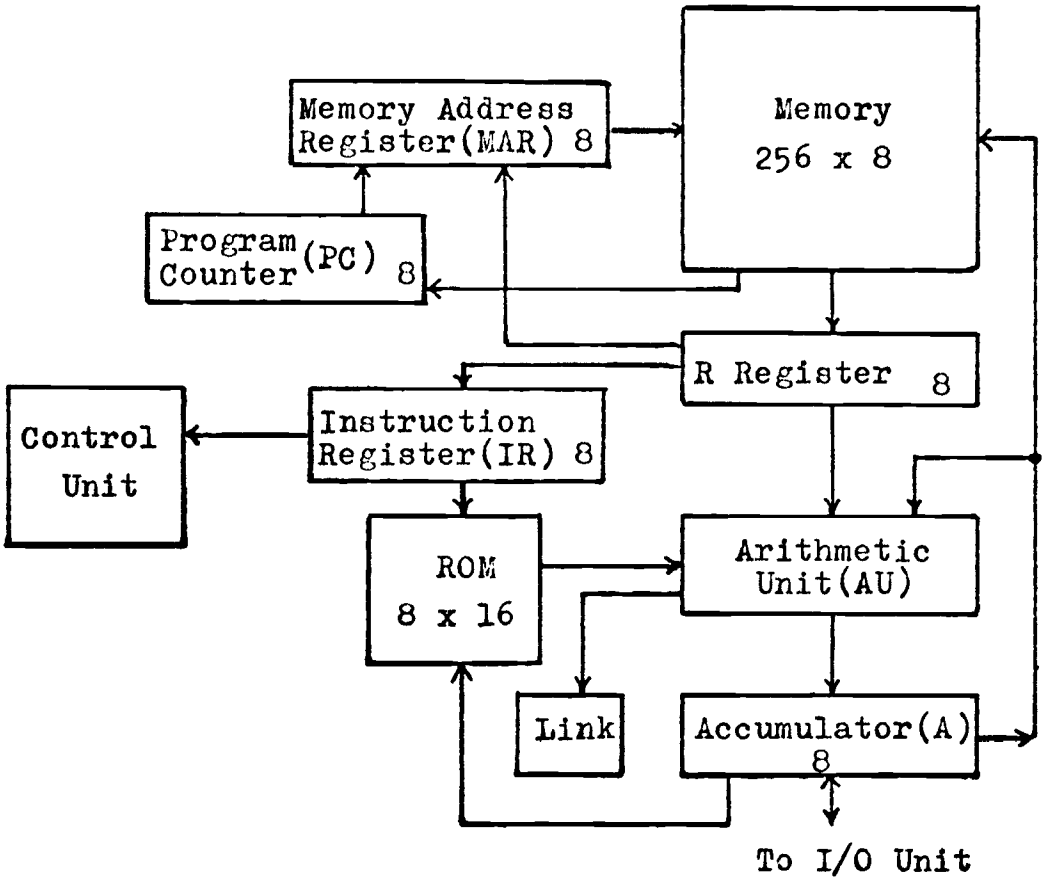


Figure 1.1 Register Organization of the Computer.

could be used as a function-select matrix in place of the ROM; the only difference is that ROM stores information in a permanent form while the contents in RAM can be altered. ROM operates faster in speed and costs less than RAM. Both form of memory will be considered.

The performance of the AU will be examined in the next chapter. Structural organization of the computer is

essentially conventional but it could also be varied to include microprogramming. This will be discussed in Chapter III in connection with the operation of the computer. The logic design of each component part of the machine is presented in Chapter IV. Chapter V will be the conclusion.

II. ARITHMETIC UNIT

The Arithmetic Unit(AU) is that part of the computer performing arithmetic and logic operations. Other operations such as increment, clear, shift, etc. are usually performed by adding control gates which tend to complicate logic design as the number of these instructions increases. By employing ULBs and ROM in the AU, several instructions can actually be realized without complicated structure.

The register organization of the parallel AU is shown in Figure 2.1. There are eight cells in the AU, each has three inputs and two outputs. Each cell in the AU is constructed of a ULB shown in Figure 2.2(a). R and A are corresponding bits in the R register and the accumulator; C is the carry-in connected to the carry-out from the next least significant cell. The carry of the least significant cell is to be specified as logic 0. The gates representing minterms of the ULB are controlled by the k's directly mapped into the outputs of the ROM shown in Figure 2.2(b).

To perform an instruction, the Instruction register addresses a word location in the ROM whose contents is a pattern of 1's and 0's representing the function of the instruction. Contents of the addressed word provide the k's used as inputs to the AU. A gate is then enabled if the corresponding k_i is a 1 and disabled if it is 0. The result

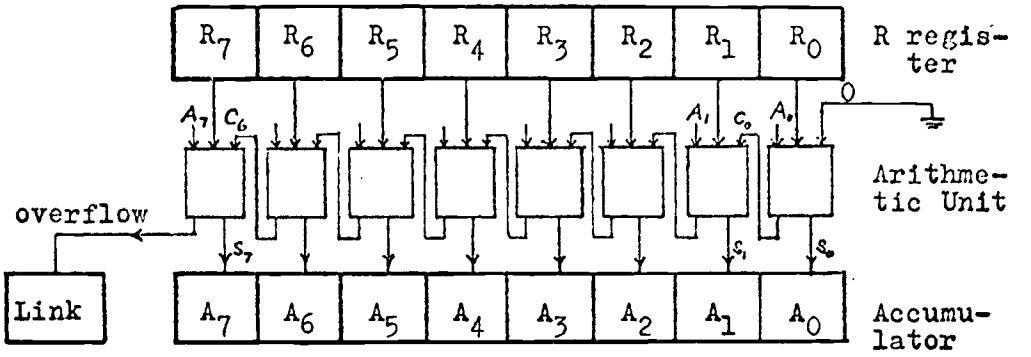


Figure 2.1 Register Organization of the AU.

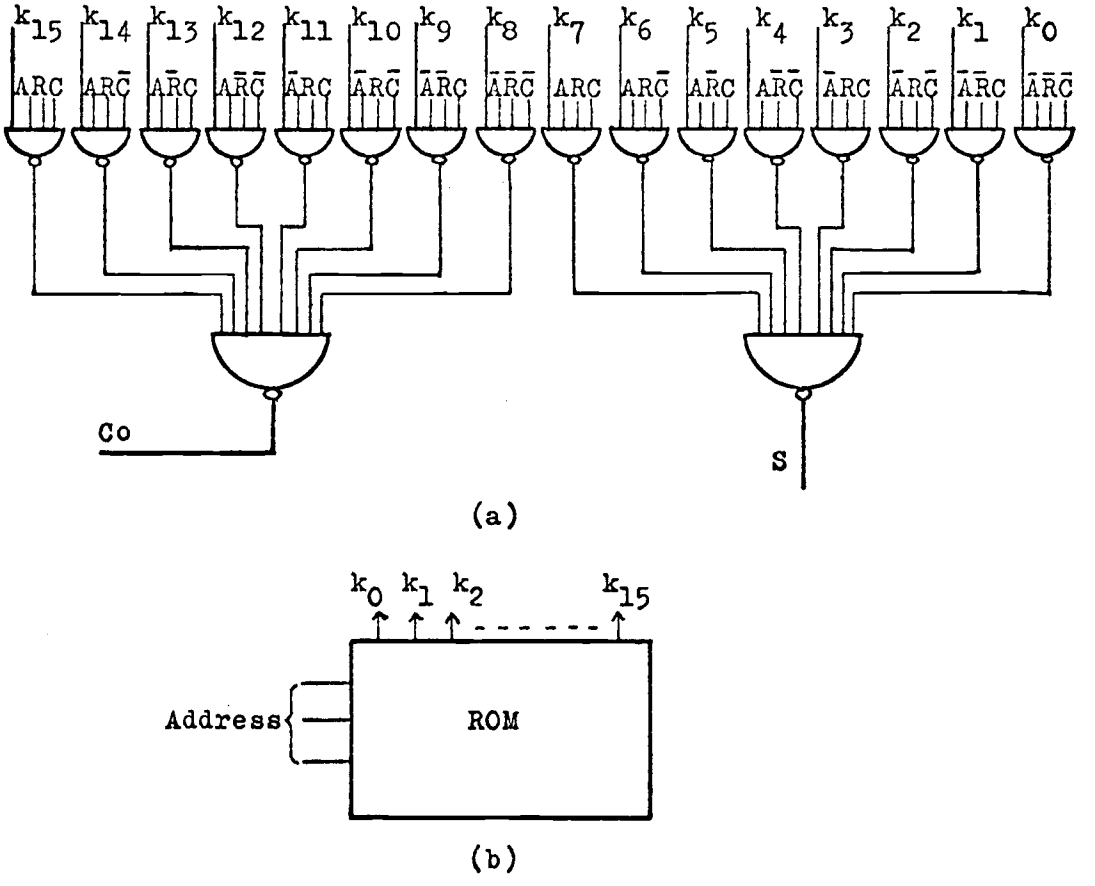


Figure 2.2 (a) One Cell of the AU; (b) The ROM.

of the operation given by S is normally put back in the accumulator.

The binary information stored in the ROM is the truth table of the desired functions. Addition for example has a truth table for the sum bit and the carry bit as shown.

ADD

A	R	C	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\underline{K} = (\overset{S}{0\ 1\ 1\ 0\ 1\ 0\ 0\ 1}\ \ \overset{Co}{0\ 0\ 0\ 1\ 0\ 1\ 1\ 1})$$
$$k_0 k_1 \ldots\ldots\ldots k_7\ k_8\ \ldots\ldots\ldots k_{15}$$

Information to be stored in the ROM is therefore its truth value in columns S and Co arranged in the order indicated by K. When addition is performed, only minterms corresponding to a 1 under column S are selected to compute the sum and those under Co are selected to generate the carry.

Subtraction and the 16 logical operations are obtained in a similar manner. The "carry" input becomes a borrow. Other functions are possible utilizing the carry-in of the least significant cell which is a logic 0. For example, if a binary number stored in A is to be increased by 1, it can be arbitrarily assumed that $\underline{C}=0$ means a carry

has been generated. Hence in the following table, S is computed by adding a 1 to A if C=0 and adding a 0 if C=1. Similarly, if a carry is generated in computing the sum, a 0 is entered under column Co.

INCREMENT

A	R	C	S	Co
0	0	0	1	1
0	0	1	0	1
0	1	0	1	1
0	1	1	0	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

$K=$ (1 0 1 0 0 1 0 1 1 1 1 1 0 1 0 1)
S Co
$k_0 k_1 \ldots\ldots\ldots\ldots\ldots\ldots k_{15}$

The parity of information stored in A can be performed by the propagation of the carry and detecting the overflow in the Link register. Assume for example the carry-in, C=0, of the least significant cell means an even parity check. This carry will propagate through each cell changing its value alternatively whenever a position in A is a 1. At the end of the operation, if the parity check of the digits in A shows an odd parity, the Link will be set to 1, otherwise Link=0. The function of an odd parity check is shown below.

ODD PARITY CHECK

A	R	C	S	Co
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	0

K= (0 0 0 0 1 1 1 1 0 1 0 1 1 0 1 0)
(S Co)
k_0 ..... k_{15}

Instruction to determine the status of specific digits in the accumulator can also be performed through masking by the R register. For example if

R= 00110011
and A= 11011011 ,

an instruction called "Odd parity check on mask" when performed will produce an overflow(Link=1) since there are odd number of 1's in the position of A for which the corresponding R bit is 1. The function of this instruction, illustrated in the following table, is obtained by assuming the

ODD PARITY CHECK
ON MASK

A	R	C	S	Co
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

K= (0 0 0 0 1 1 1 1 0 1 0 1 0 1 1 0)
(S Co)
k_0 ..... k_{15}

carry C=0 to mean an even parity check. Therefore when both A and R = 1, Co will have value opposite to C indicating a change of parity has been observed. The values in column S should be the same as column A so that the operation will not destroy contents of the A register.

Table 2.1 lists the characteristics of several representative instructions. They are described in the following pages.

Table 2.1 Instructions

Arguments \ Functions			ADD		SUBTRACT		COMPLEMENT		NAND	
A	R	C	S	Co	S	Co	S	Co	S	Co
0	0	0	0	0	0	0	1	d	1	d
0	0	1	1	0	1	1	1	d	1	d
0	1	0	1	0	1	1	1	d	1	d
0	1	1	0	1	0	1	1	d	1	d
1	0	0	1	0	1	0	0	d	1	d
1	0	1	0	1	0	0	0	d	1	d
1	1	0	0	1	0	0	0	d	0	d
1	1	1	1	1	1	1	0	d	0	d
Typical cell										
	A		augend		minuend		operand		1st operand	
	R		addend		subtrahend		not used		2nd operand	
	C=	0	no carry		no borrow		don't care		don't care	
		1	carry		borrow		"		"	
	Link=	0	no overflow		no underflow		"		"	
		1	overflow		underflow		"		"	
	S		sum		difference		$\overline{A}$		A NAND R	
	K		(01101001 00010111)		(01101001 01110001)		(11110000 dddddddd)		(11111100 dddddddd)	

Arguments \ Functions			NOR		AND		OR		Ex-OR	
A	R	C	S	Co	S	Co	S	Co	S	Co
0	0	0	1	d	0	d	0	d	0	d
0	0	1	1	d	0	d	0	d	0	d
0	1	0	0	d	0	d	1	d	1	d
0	1	1	0	d	0	d	1	d	1	d
1	0	0	0	d	0	d	1	d	1	d
1	0	1	0	d	0	d	1	d	1	d
1	1	0	0	d	1	d	1	d	0	d
1	1	1	0	d	1	d	1	d	0	d
Typical cell										
	A		1st operand		1st operand		1st operand		1st operand	
	R		2nd operand		2nd operand		2nd operand		2nd operand	
	C=	0	don't care		don't care		don't care		don't care	
		1	"		"		"		"	
	Link=	0	"		"		"		"	
		1	"		"		"		"	
	S		A NOR R		A AND R		A OR R		A $\oplus$ R	
	K		(11000000 dddddddd)		(00000011 dddddddd)		(00111111 dddddddd)		(00111100 dddddddd)	

Arguments \ Functions			EQUAL		NO-OP		SHIFT LEFT		INCREMENT	
A	R	C	S	Co	S	Co	S	Co	S	Co
0	0	0	1	d	0	d	0	0	1	1
0	0	1	1	d	0	d	1	0	0	1
0	1	0	0	d	0	d	0	0	1	1
0	1	1	0	d	0	d	1	0	0	1
1	0	0	0	d	1	d	0	1	0	0
1	0	1	0	d	1	d	1	1	1	1
1	1	0	1	d	1	d	0	1	0	0
1	1	1	1	d	1	d	1	1	1	1

Typical cell				
A	1st operand	operand	operand	operand
R	2nd operand	not used	not used	not used
C= 0	don't care	don't care	A_{i-1}	carry
1	"	"		no carry
Link= 0	"	"	A_{i-1}	overflow
1	"	"	$\bar{A}_{i-1}$	no overflow
S	A EQUAL R	A	$\bar{A}_{i-1}$	A + 1
K	(11000011 dddddddd)	(00001111 dddddddd)	(01010101 00001111)	(10100101 11110101)

Arguments \ Functions			DECREMENT		TWO'S COMPLEMENT		A + R + 1		A - R - 1	
A	R	C	S	Co	S	Co	S	Co	S	Co
0	0	0	1	0	0	0	1	1	1	0
0	0	1	0	1	1	1	0	1	0	1
0	1	0	1	0	0	0	0	0	0	0
0	1	1	0	1	1	1	1	1	1	0
1	0	0	0	1	1	1	0	0	0	1
1	0	1	1	1	0	1	1	1	1	1
1	1	0	0	1	1	1	1	0	1	0
1	1	1	1	1	0	1	0	0	0	1

Typical cell				
A	operand	operand	1st operand	1st operand
R	not used	not used	2nd operand	2nd operand
C= 0	borrow	carry	carry	borrow
1	no borrow	no carry	no carry	no borrow
Link= 0	underflow	overflow	overflow	underflow
1	no "	no "	no "	no "
S	A - 1	$\bar{A}$ + 1	A + R + 1	A - R - 1
K	(10100101 01011111)	(01011010 01011111)	(10010110 11010100)	(10010110 01001101)

Arguments \ Functions	A + A + 1	Clear A	Increment A on mask	Decrement A on mask
A R C	S Co	S Co	S Co	S Co
0 0 0	1 1	0 d	0 0	0 0
0 0 1	0 1	0 d	0 1	0 1
0 1 0	1 1	0 d	1 1	1 0
0 1 1	0 1	0 d	0 1	0 1
1 0 0	1 0	0 d	1 0	1 0
1 0 1	0 0	0 d	1 1	1 1
1 1 0	1 0	0 d	0 0	0 1
1 1 1	0 0	0 d	1 1	1 1
Typical cell				
A	operand	operand	operand	operand
R	not used	not used	mask	mask
C= 0	carry	don't care	carry	borrow
1	no carry	"	no carry	no borrow
Link= 0	overflow	"	overflow	underflow
1	no "	"	no "	no "
S	A + A + 1	0	masked A+1	masked A-1
K	(10101010 11110000)	(00000000 dddddddd)	(00101101 01110101)	(00101101 01010111)

Arguments \ Functions	0's A on mask	Complement A on mask	Test for 00000000 in A	Test for 11111111 in A
A R C	S Co	S Co	S Co	S Co
0 0 0	0 d	0 d	0 0	0 1
0 0 1	0 d	0 d	0 1	0 1
0 1 0	0 d	1 d	0 0	0 1
0 1 1	0 d	1 d	0 1	0 1
1 0 0	1 d	1 d	1 1	1 0
1 0 1	1 d	1 d	1 1	1 1
1 1 0	0 d	0 d	1 1	1 0
1 1 1	0 d	0 d	1 1	1 1
Typical cell				
A	operand	operand	operand	operand
R	mask	mask	not used	not used
C= 0	don't care	don't care	A is 0	A is 1
1	"	"	A is 1	A is 0
Link= 0	"	"	A is 0	A is 1
1	"	"	A is 1	A is 0
S	masked A-0	masked A-$\bar{A}$	A	A
K	(00001100 dddddddd)	(00111100 dddddddd)	(00001111 01011111)	(00001111 11110101)

Arguments \ Functions	Odd parity check	Odd parity check on mask	R>A	R<A
A R C	S Co	S Co	S Co	S Co
0 0 0	0 0	0 0	0 0	0 0
0 0 1	0 1	0 1	0 1	0 1
0 1 0	0 0	0 0	0 1	0 0
0 1 1	0 1	0 1	0 1	0 0
1 0 0	1 1	1 0	1 0	1 1
1 0 1	1 0	1 1	1 0	1 1
1 1 0	1 1	1 1	1 0	1 0
1 1 1	1 0	1 0	1 1	1 1
Typical cell				
A	operand	operand	1st operand	1st operand
R	not used	mask	2nd operand	2nd operand
C= 0	even parity	even parity	R≠A	R≥A
1	odd "	odd "	R>A	R<A
Link= 0	even "	even "	R≠A	R≥A
1	odd "	odd "	R>A	R<A
S	A	A	A	A
K	(00001111 01011010)	(00001111 01010110)	(00001111 01110001)	(00001111 01001101)

Arguments \ Functions	R≠A	Load	Load Two's Complement	Odd parity check in R
A R C	S Co	S Co	S Co	S Co
0 0 0	0 0	0 d	0 0	0 0
0 0 1	0 1	0 d	1 1	0 1
0 1 0	0 1	1 d	1 1	1 1
0 1 1	0 1	1 d	0 1	1 0
1 0 0	1 1	0 d	0 0	0 0
1 0 1	1 1	0 d	1 1	0 1
1 1 0	1 0	1 d	1 1	1 1
1 1 1	1 1	1 d	0 1	1 0
Typical cell				
A	1st operand	not used	not used	not used
R	2nd operand	operand	operand	operand
C= 0	R=A	don't care	carry	even parity
1	R≠A	"	no carry	odd "
Link= 0	R=A	"	overflow	even "
1	R≠A	"	no "	odd "
S	A	R	$\overline{R}$+1	R
K	(00001111 01111101)	(00110011 dddddddd)	(01100110 01110111)	(00110011 01100110)

III. ORGANIZATION AND OPERATION

The preceding chapter has shown the operation of the AU in realizing many instructions. We now apply the unit into a small computer to test out its feasibility and examine the characteristics of the machine.

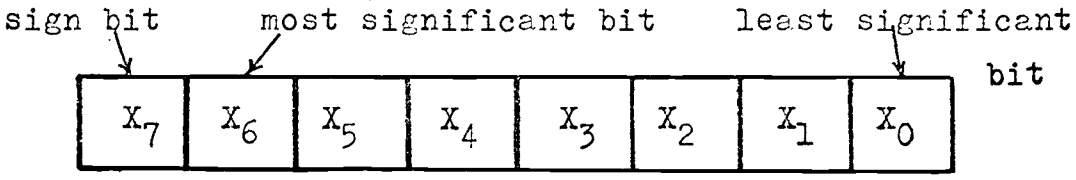
Organization

The computer is to be a general-purpose stored program digital computer. It will be specified to operate in parallel, synchronous, single address instruction type, and fixed word length. The word format is illustrated in Figure 3.1. Numbers are to be binary integers in 2's complement representation. For commands requiring access of the operands from the memory, three bits are used to specify the operation and five bits are to specify the address. For commands requiring no access to the memory, eight bits could be used but seven bits will be enough to specify the operation. In the I/O instruction format, the left four bits represent the operation code and the remaining bits are used to specify the external device code.

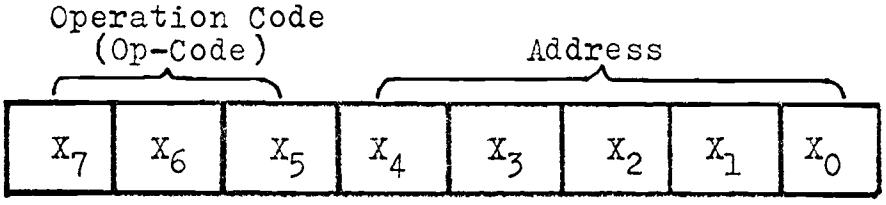
This computer will use semi-conductor memory in the storage unit to store data and instructions. The registers are to be constructed of J-K flip-flops. The use of each register, referring back to Figurel.1, are outlined below:

MAR- Specifies a word location in the memory.

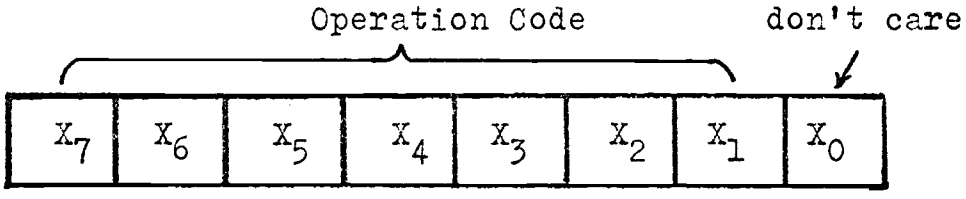
R- Holds information removed from the memory.



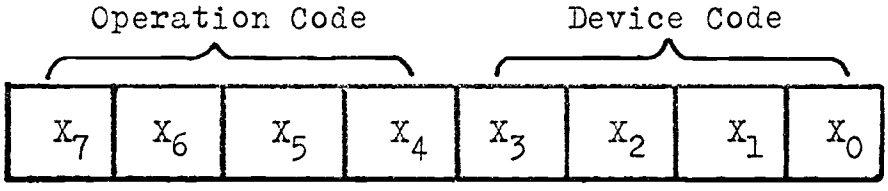
(a) Data Format



(b) Memory Access Instruction Format



(c) Non-Memory Access Instruction Format



(d) I/O Instruction Format

Figure 3.1 The Word Format.

A- Stores the result of an operation performed by the AU, it also serves as an input-output register.

PC- Holds the memory address of each subsequent instruction.

IR- Holds an instruction word, addresses the ROM, and its contents is presented to a decoding circuit.

The commands to be executed by the computer will be divided into four classes:

Class(11) are commands requiring access to both the memory and ROM. For example an ADD command needs the operand in the memory and the appropriate K value from the ROM.

Class(10) are commands requiring access to the memory only. For example to STORE a word into the memory requires address of the location to be stored.

Class(01) are commands requiring access to the ROM only, such as to CLEAR the accumulator.

Class(00) are commands requiring no access to either memory. For example the HALT instruction.

Operation

Prior to operation, instructions and data must first be loaded into the memory either through the use of switches or by means of an I/O device. The basic sequence of operation consists of an alternation of a time period called the Instruction Cycle, followed by a period of time called the Execution Cycle. The flow of information during each cycle is described in the following.

Instruction Cycle. At the start of the cycle, an instruction word in a memory location addressed by the MAR is removed and placed into the R register. At the next clock

pulse, three mutually exclusive operations occur: the address portion of R is sent to the MAR giving the address of the operand to be used; the whole contents of R is transfered to the IR for interpretation by the control unit; and PC is increased by 1 providing the address of the next instruction. Each instruction to be executed will require the same operations stated above.

Execution Cycle. In the execution cycle, an operand in a memory location whose address is now specified by the MAR is removed and copied into the R register. Except for the Class(11) and Class(10) commands, this operand will be meaningless. At the next clock pulse, the operation specified by the instruction word during the instruction cycle is performed. Since the IR will be constantly addressing the ROM, the AU may be operating upon a meaningless operand. The result of the operation may or may not be stored in the accumulator depending on the class of the command.

The cycles are repeated for each instruction in a program until the program comes to a halt.

The role of ROM in the computer may be divided into two classes:

(1) The size of ROM has been chosen to be 8 words by 16 bits, obviously all functions listed in Table 2.1 cannot be accomodated. In case some functions are required but were not stored in ROM, two procedures are possible without

making any changes in the circuits.

a. The ROM is taken out and plugged in with a ROM with different contents.

b. A RAM may be used instead of ROM. This allows functions stored in the RAM to be altered under program control.

In this case each location in the function-select memory corresponds to a single command from the main memory.

(2) When an instruction in a program calls for repeated execution of some commands, it is possible to install a ROM using microprogramming. Each instruction from the main memory corresponds to several sequential instructions in the ROM. This eliminates the need to unnecessarily repeat the instruction cycle.

An example for multiplication is illustrated in Figure 3.2. The three bits of IR which was used before to address the eight word locations in the ROM is now used to

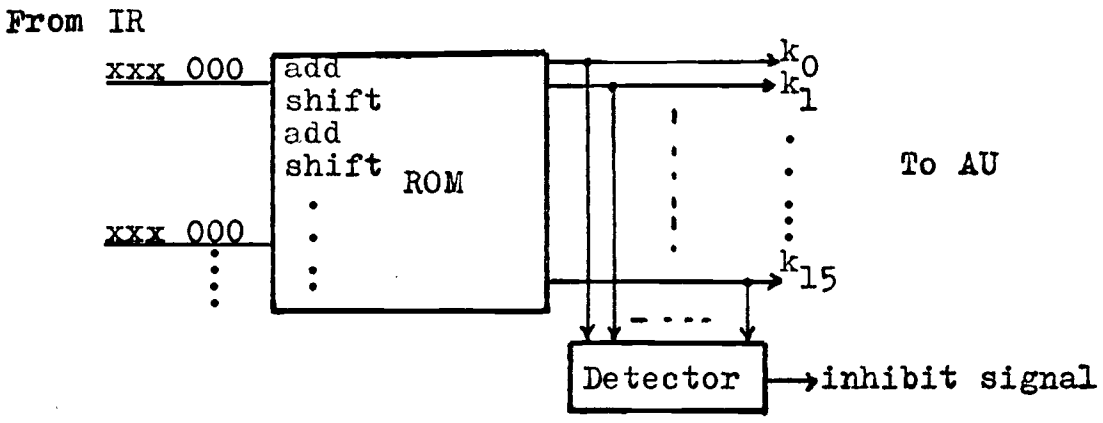


Figure 3.2 ROM used for Microprogramming.

specify the three most significant digits in the address of a larger ROM. Operations start in an address with 0's in the less significant digits and execute down consecutively until the detector senses an inhibit code which terminates the microprogram. The limitation however is that the operands must be in the proper registers.

The commands to be executed by the computer are listed in Table 3.1. The op-code in Class(11) and the three bits underlined in the op-code of Class (01) commands are to specify the eight addresses in the ROM. The contents in each word location will be the binary function of the corresponding instruction. Description of the symbols adopted in the table are:

( )	Contents of a register. Ad(R) means contents of the address part of an instruction in R register.
M⟨MAR⟩	Contents of a memory word whose address is specified by the MAR.
⟶	Means information is transfered to.

The operations performed during the instruction cycle are:

$$\begin{cases} (1) & M\langle MAR\rangle \rightarrow R \\ (2) & Ad(R) \rightarrow MAR \\ & (R) \rightarrow IR \\ & (PC)+1 \rightarrow PC \end{cases}$$

Where (1) and (2) denote the subcycles.

Table 3.1 List of Instructions

Op-Code	Class	Commands	Mnemonic	Execution Cycle
100xxxxx	11	Add	ADD	(1) M⟨MAR⟩→R (2) (R) + (A)→A (PC)→MAR
101xxxxx		Subtract	SUB	(1) M⟨MAR⟩→R (2) (A) - (R)→A (PC)→MAR
110xxxxx		AND	AND	(1) M⟨MAR⟩→R (2) (R)AND(A)→A (PC)→MAR
111xxxxx		OR	ORR	(1) M⟨MAR⟩→R (2) (R)OR(A)→A (PC)→MAR
001xxxxx	10	Store	STO	(1) (A)→M⟨MAR⟩ (2) (PC)→MAR
010xxxxx		Jump (indirect)	JMI	(1) M⟨MAR⟩→PC (2) (PC)→MAR
0000000d	01	Increment	INC	(1) (A) + 1→A (PC)→MAR
0000010d		Shift Left	SIF	(1) $A_{i-1} \rightarrow A_i$ (PC)→MAR
0000100d		Clear	CLR	(1) 0→A (PC)→MAR
0000110d		Complement	COM	(1) $\bar{A}$→A (PC)→MAR
011xxx0d		Store Left	LEF	(1) (A)→RAM⟨IR⟩ (2) (PC)→MAR
011xxx1d		Store Right	RIT	(1) (A)→RAM⟨IR⟩ (2) (PC)→MAR
0000001d	00	Branch on Negative	BRN	if A7 = 0 (1) (PC)→MAR if A7 = 1 (1) (PC) + 1→PC (2) (PC)→MAR

continue..

Op-Code	Class	Command	Mnemonic	Execution Cycle
0000011d	00	Branch on Link	BRL	(1) if Link= 0 (PC)→MAR (1) if Link= 1 (PC)+1→PC (2) (PC)→MAR
0000101d		No Operation	NOP	
0000111d		Halt	HLT	
0001xxxx	I/O	I/O Instructions		(1) I/O instruction is executed (2) (PC)→MAR

Three programs will be shown to illustrate the operation of the computer. The first program is to add two numbers stored in locations 20 and 21 in the memory. The second program is to change the function of an instruction stored in location 5 in the RAM(function-select) to the instruction "Odd parity check". The third is a hypothetical microprogram utilizing ROM for multiplication subroutine. The multiplicand is stored in location 4 in the main memory. Because this computer does not have an MQ register to store the multiplier, we will assume the multiplier is a constant number 15, already stored in the ROM. We will also assume there are logic circuits in the computer to resolve the consecutive addressing of the ROM and that the overflow problem is eliminated.

PROGRAM I

Location	Contents	Descriptions
0	00001000	Clear the accumulator
1	10010100	Add contents of location 20
2	10010101	Add contents of location 21
3	00001111	Halt
4 ⋮ 19	not used	
20	00001011	Data 11
21	11111100	Data - 4

RUN

	PC	MAR	IR	R	A
I-cycle0	00000001	00001000	00001000	00001000	xxxxxxxx
E-cycle0	00000001	00000001	00001000	xxxxxxxx	00000000
I-cycle1	00000010	00010100	10010100	10010100	00000000
E-cycle1	00000010	00000010	10010100	00001011	00001011
I-cycle2	00000011	00010101	10010101	10010101	00001011
E-cycle2	00000011	00000011	10010101	11111100	00000111
I-cycle3	00000100	00001111	00001111	00001111	00000111
E-cycle3	00000100	00000100	00001111	xxxxxxxx	00000111

The x's are meaningless contents.

PROGRAM II

Location	Contents	Descriptions
0	00001000	Clear the accumulator
1	10001010	Add contents of location 10
2	01110100	Store left the contents of A
3	00001000	Clear the accumulator
4	10001011	Add contents of location 11
5	01110111	Store right the contents of A
6	00001111	Halt
7, 8, 9	not used	
10	00001111	Binary function from k0 to k7
11	01011010	Binary function from k8 to k15

RUN

	PC	MAR	IR	R	A
I-cycle0	00000001	00001000	00001000	00001000	xxxxxxxx
E-cycle0	00000001	00000001	00001000	xxxxxxxx	00000000
I-cycle1	00000010	00001010	10001010	10001010	00000000
E-cycle1	00000010	00000010	10001010	00001111	00001111
I-cycle2	00000011	00010100	01110100	01110100	00001111
E-cycle2	00000011	00000011	01110100	xxxxxxxx	00001111
I-cycle3	00000100	00001000	00001000	00001000	00001111
E-cycle3	00000100	00000100	00001000	xxxxxxxx	00000000
I-cycle4	00000101	00001011	10001011	10001011	00000000
E-cycle4	00000101	00000101	10001011	01011010	01011010
I-cycle5	00000110	00010111	01110111	01110111	01011010
E-cycle5	00000110	00000110	01110111	xxxxxxxx	01011010
I-cycle6	00000111	00001111	00001111	00001111	01011010
E-cycle6	00000111	00000111	00001111	xxxxxxxx	01011010

The x's are meaningless contents.

Program III

Location	Contents	Descriptions
0	00001000	Clear the accumulator
1	11100100	Multiply contents of location 4, assuming the op-code is 111zzzzz.
2	00001111	
3	not used	
4	00000011	Data 3

RUN

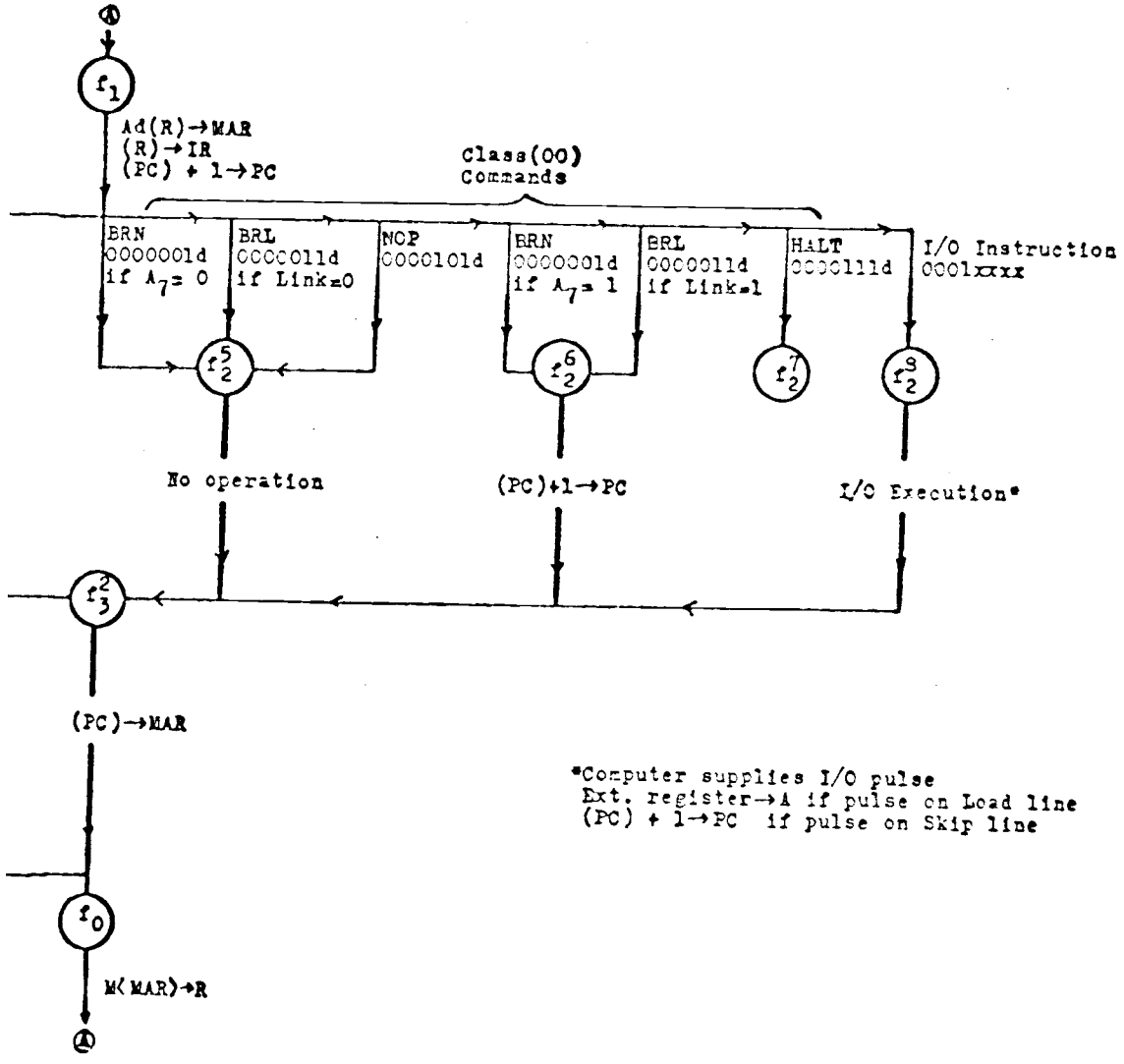
	PC	MAR	IR	R	A	
I-c0	00000001	00001000	00001000	00001000	xxxxxxxx	
E-c0	00000001	00000001	00001000	xxxxxxxx	00000000	
I-c1	00000010	00000100	11100100	11100100	00000000	
E-c(1)	00000010	00000010	11100100	00000011	00000011	Add
(2)	00000010	00000010	11100100	00000011	00000110	Shift
(3)	00000010	00000010	11100100	00000011	00001001	Add
(4)	00000010	00000010	11100100	00000011	00010010	Shift
(5)	00000010	00000010	11100100	00000011	00010101	Add
(6)	00000010	00000010	11100100	00000011	00101010	Shift
(7)	00000010	00000010	11100100	00000011	00101101	Add
(8)	INHIBIT CODE DETECTED					
I-c2	00000011	00001111	00001111	00001111	00101101	
E-c2	00000011	00000011	00001111	xxxxxxxx	00101101	

The x's are meaningless contents.

IV. LOGIC DESIGN

This chapter presents the design of logic circuits of the computer to implement the prescribed operations worked out previously.

The Control Unit. The control unit consists of an operation counter and decoding circuitry. The counter controls the sequential operation of the computer while the decoding circuits decode machine instructions into command signals. The sequential operation of each instruction listed in Table 3.1 is described in a state diagram shown in Figure 4.1. The states denoted by f's in the state diagram correspond to the states of the operation counter. Each state is assigned specific operations to be performed in a period of one clock-pulse time. The instruction cycle is assigned to states f_0 and f_1 while the execution cycle to states f_2^i and f_3^i, where i denotes the command in that state. Since the execution of any instruction would go through four states at most, a minimum of two-stage counter is required to sequence the state transition. The counter is designed by construction of a transition table and the state assignment as shown in Table 4.1.



of the Computer

Table 4.1 The Transition Table and State Assignment.

State Assignment	Present State Q_0 Q_1	Next State Q_0 Q_1	J-K Flip-Flops J_0 K_0 J_1 K_1
$f_0 = 00$	0 0	0 1	0 d 1 d
$f_1 = 01$	0 1	1 0	1 d d 1
$f_2 = 10$	1 0	1 1	d 0 1 d
$f_3 = 11$	1 1	0 0	d 1 d 1

The logic equations obtained from the table are:

$$J_0 = Q_1 \qquad J_1 = 1$$
$$K_0 = Q_1 \qquad K_1 = 1$$

Figure 4.2 shows the logic circuitry for the operation counter. The sequencing pulses are denoted by the same f_j to correspond with the states of the state diagram. The reset button is used to clear the counter as well as all register flip-flops in the computer. Manual clock control is provided for observing operations by use of the manual button. The computer will stop soon as the clock is disabled by a HALT instruction.

The decoding circuitry is shown in Figure 4.3. All instruction requiring command signals are decoded according to their operation code.

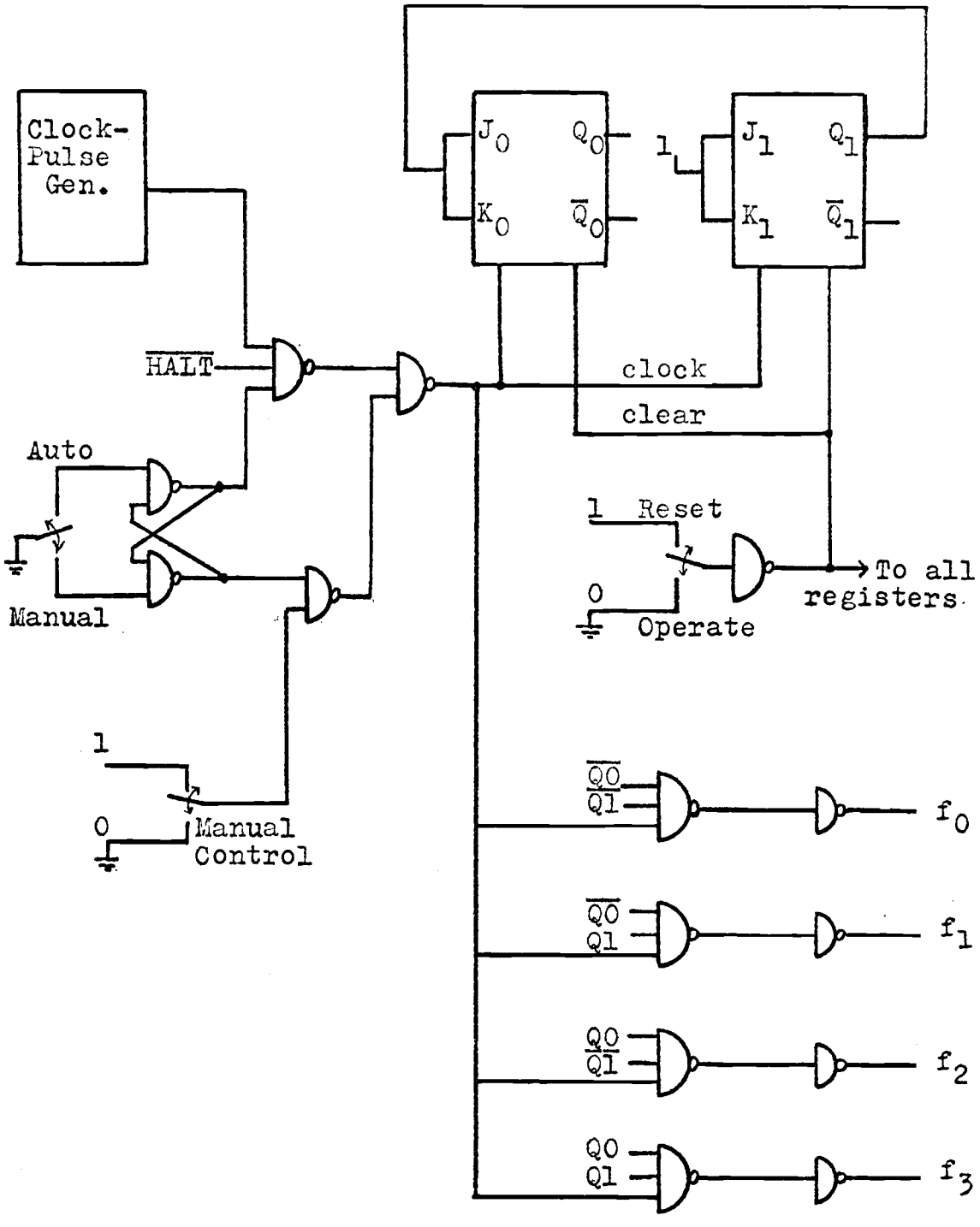


Figure 4.2 The Operation Counter.

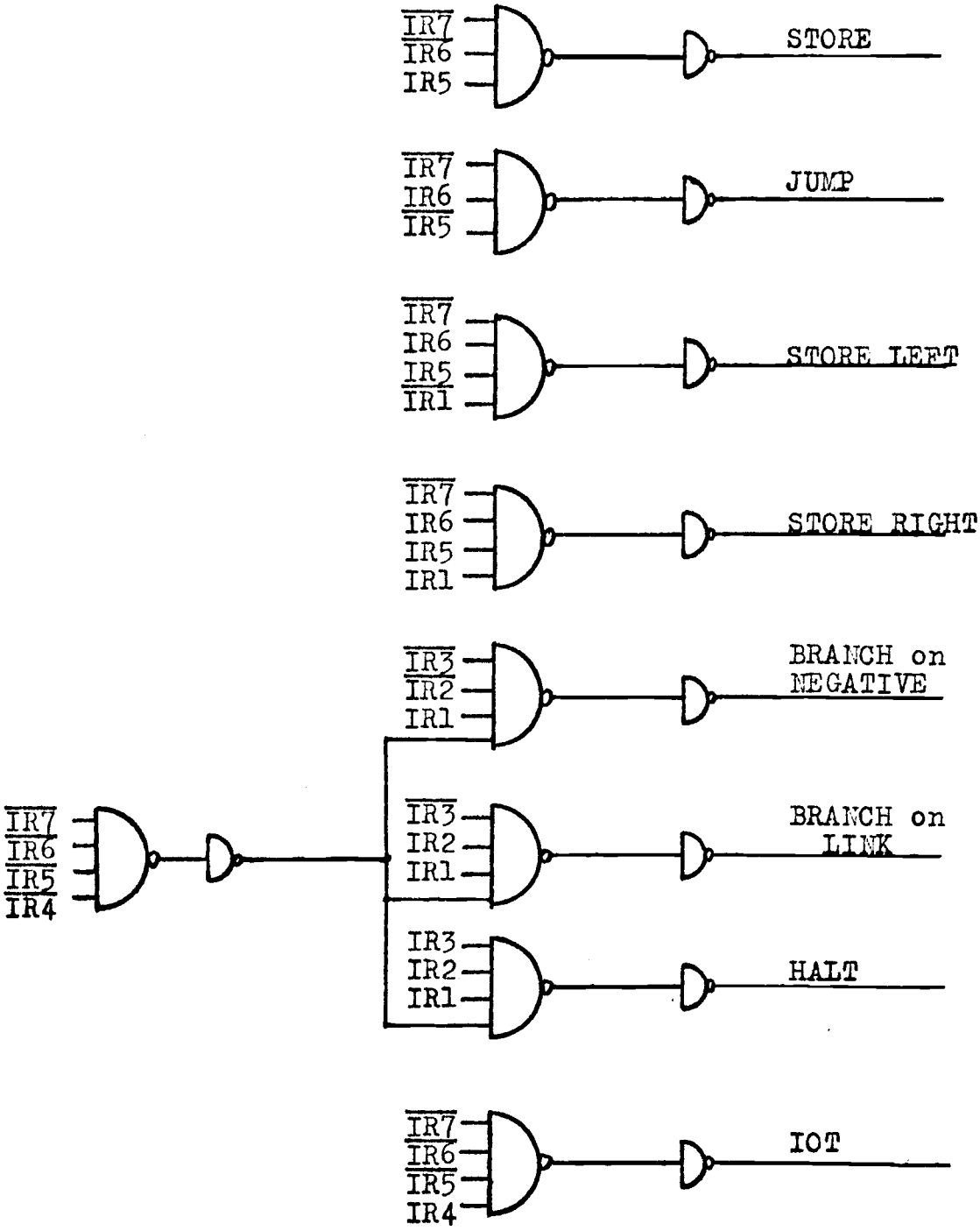


Figure 4.3 The Decoder.

I/O Data Transfer. When the decoder detects an I/O instruction, an IOP pulse will be generated during sequence f_2. This pulse will be supplied to all external devices. Each device is responsible for detecting its own select code and for providing any necessary gating. Figure 4.4 shows the information flow into and out of the computer.

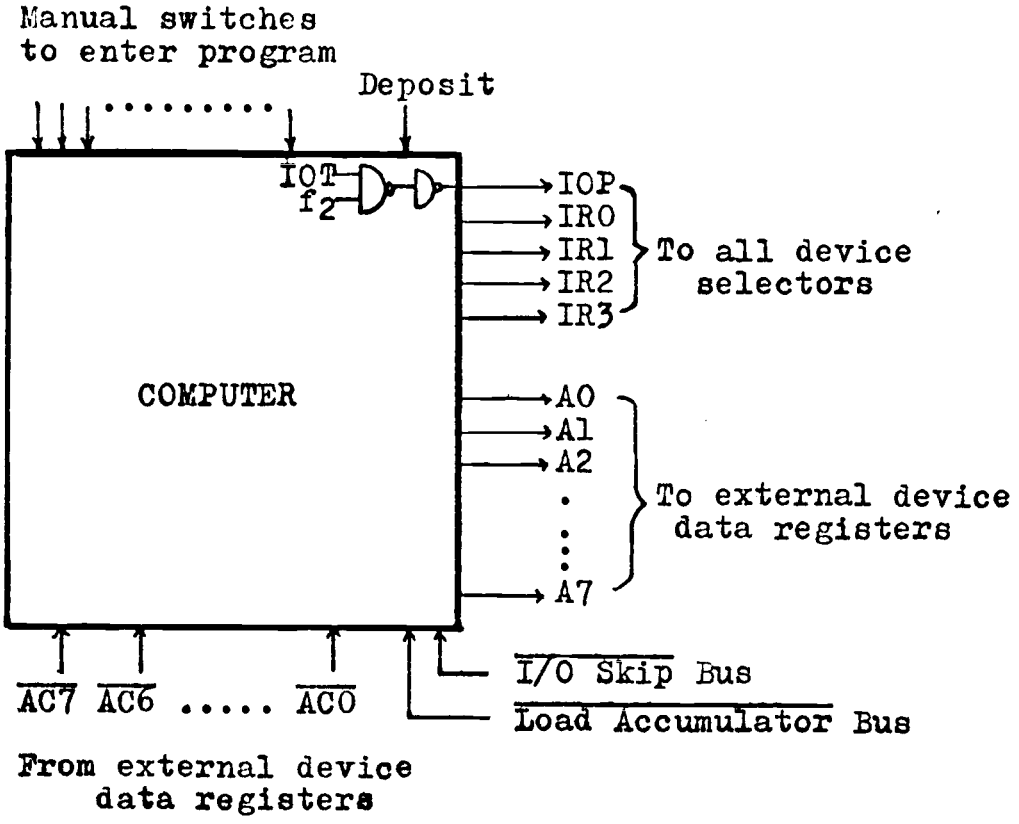


Figure 4.4 I/O Data Transfer.

The loading of data into the accumulator of the computer from an external device is illustrated in Figure 4.5. The device selector(a decoder) upon recognition of its code will activate the Load Bus and gate the contents of the data register into the accumulator.

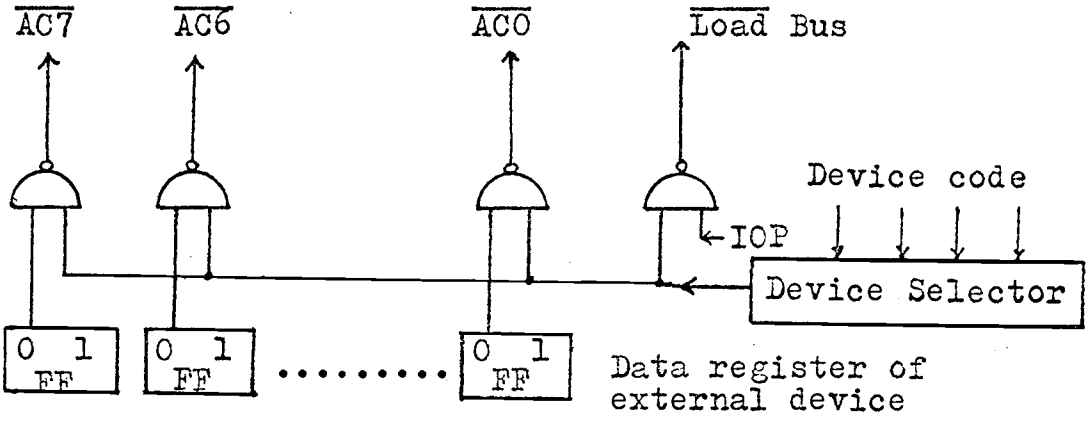


Figure 4.5 Loading Data Into the Accumulator From an External Device.

Other I/O instructions may include loading out the contents of the accumulator into an external device data register, checking, clearing, or setting a device Flag. Figures 4.6 and 4.7 illustrate the transfer of information for each of these I/O instructions.

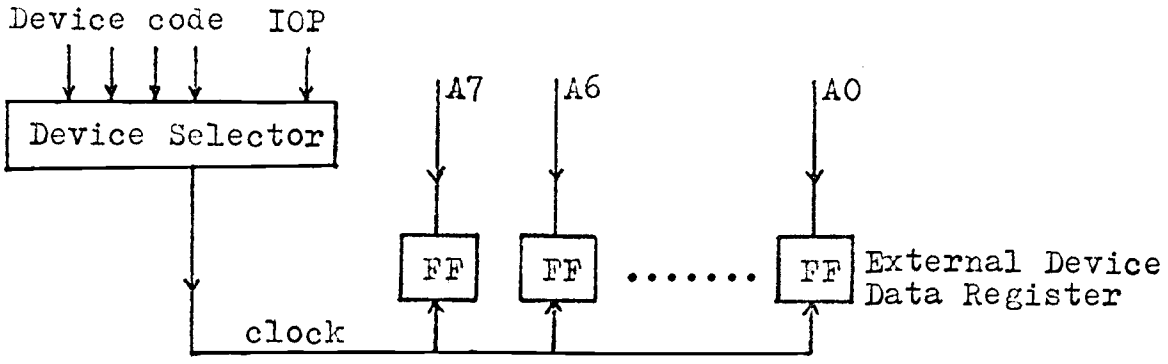


Figure 4.6 Loading Into an External Device From the Accumulator.

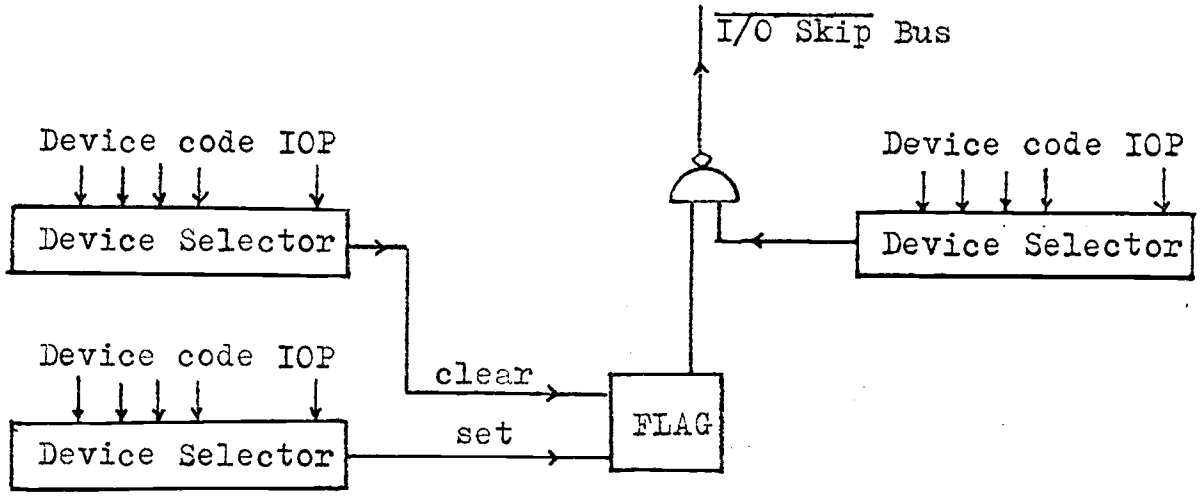


Figure 4.7 Flag of an External Device.

The I/O Skip bus is connected to the program counter in the computer. An I/O instruction to check the Flag of a device increases the PC by 1 (skip the next instruction) if the Flag=1.

A simple program to illustrate the operation of the I/O instructions is shown below. This program will load the accumulator with the contents of an external device data register. The underlined bits are the assumed device codes.

PROGRAM IV

Location	Contents	Descriptions
0	0001<u>0000</u>	Clear the Flag
1	0001<u>0001</u>	Skip on Flag
2	01001000	Jump(indirect) back to location 1
3	0001<u>0010</u>	Load data into accumulator
4	00101001	Store the contents of the accumulator into the memory in location 9
5	00001111	Halt
6, 7	not used	
8	00000001	Location 1 refered by Jump
9	xxxxxxxx	Data of the external device

The Memory Unit. The memory stores information coming from the accumulator in a location specified by MAR. The circuitry is shown in Figure 4.8. The Write Enable will be activated by a store command during sequence f_2 or by a manual Deposit button. The Deposit button loads data from two banks of switches installed in this computer(shown in Figure 4.4). Each bank consists of 8-switches. A program is manually

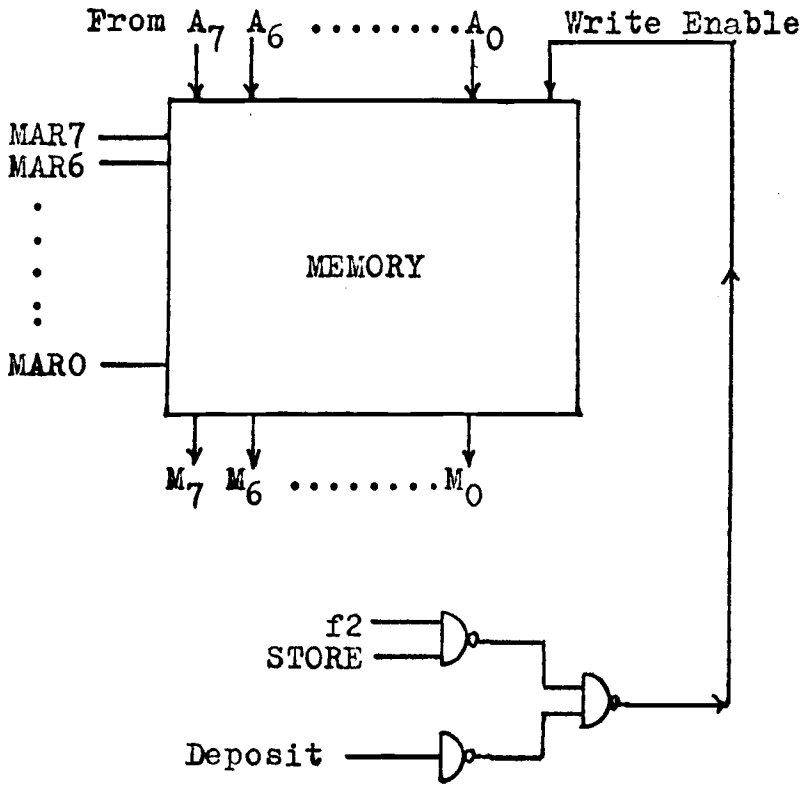


Figure 4.8 The Memory*.

*Advance Memory Systems, memory card part No. 85100511 can be used. The size is 512 Eight-bits, oversize allows for further expansion.

stored into the memory by setting the binary patterns of the address in one bank(switches denoted by SMs) and the data in the other bank(denoted by SAs), when the Deposit button is pressed information set by the switches will route through the MAR, A register, and then copied into the memory. The switches are turned off(logic 0) when the computer is in operation.

R Register. The only information to be loaded in R is from the memory, this occurs during f_0 or f_2. A typical stage of the R register is shown in Figure 4.9.

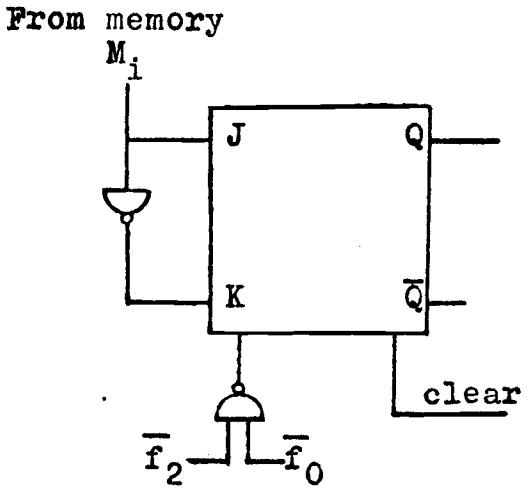


Figure 4.9 Typical Stage of the R register.

Memory Address Rgister. Referring to the state diagram, the source of information to be gated into the MAR are: the address part of R in state f_1, the PC in state f_3, and the manual SMs switches. The circuits to gate this information at the proper sequence is shown in Figure 4.10.

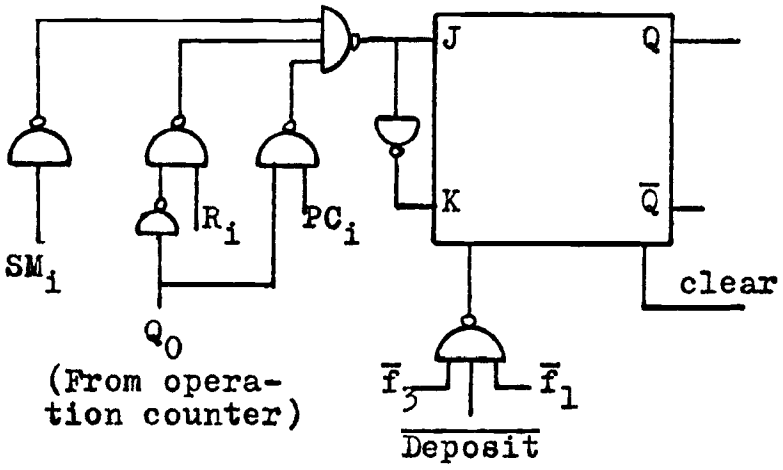
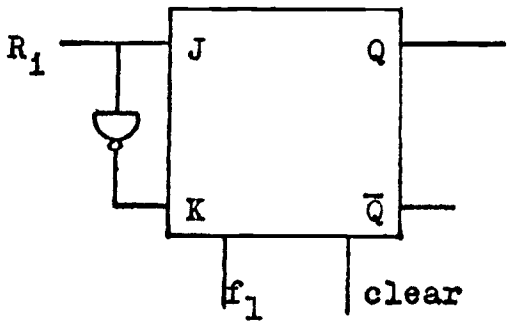
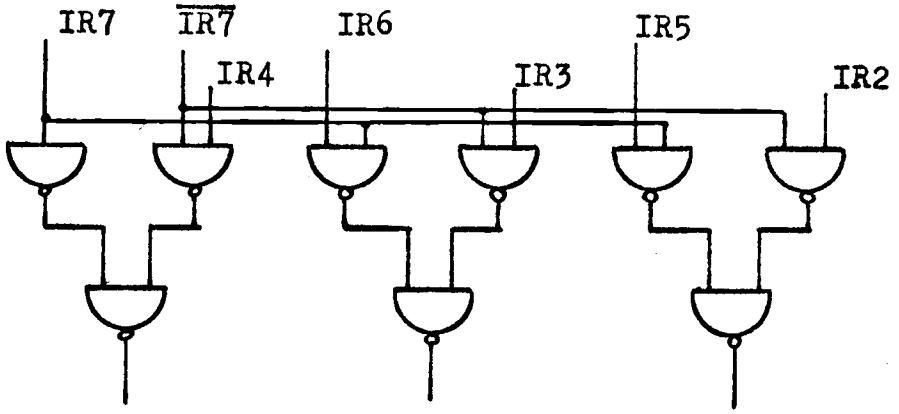


Figure 4.10 Typical Stage of the MAR.

Instruction Register. In state f_1, an instruction word from the R register is transfered to IR. The IR will then address the ROM. The contents of the IR is not altered until another instruction is executed. Figure 4,11 shows a typical stage of the IR associated with a decoder in addressing the ROM. The decoder is designed by observing the op-code of Class(11) and Class(01) commands referring



(a)

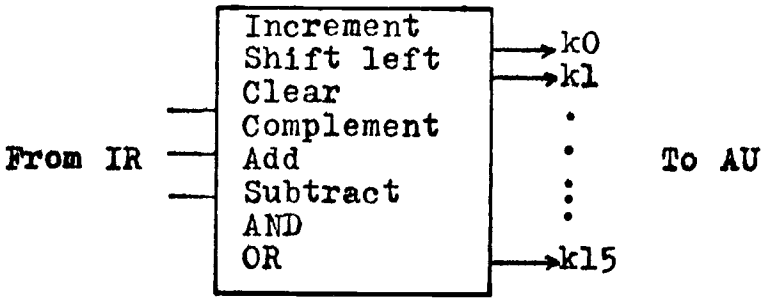


To the address of ROM

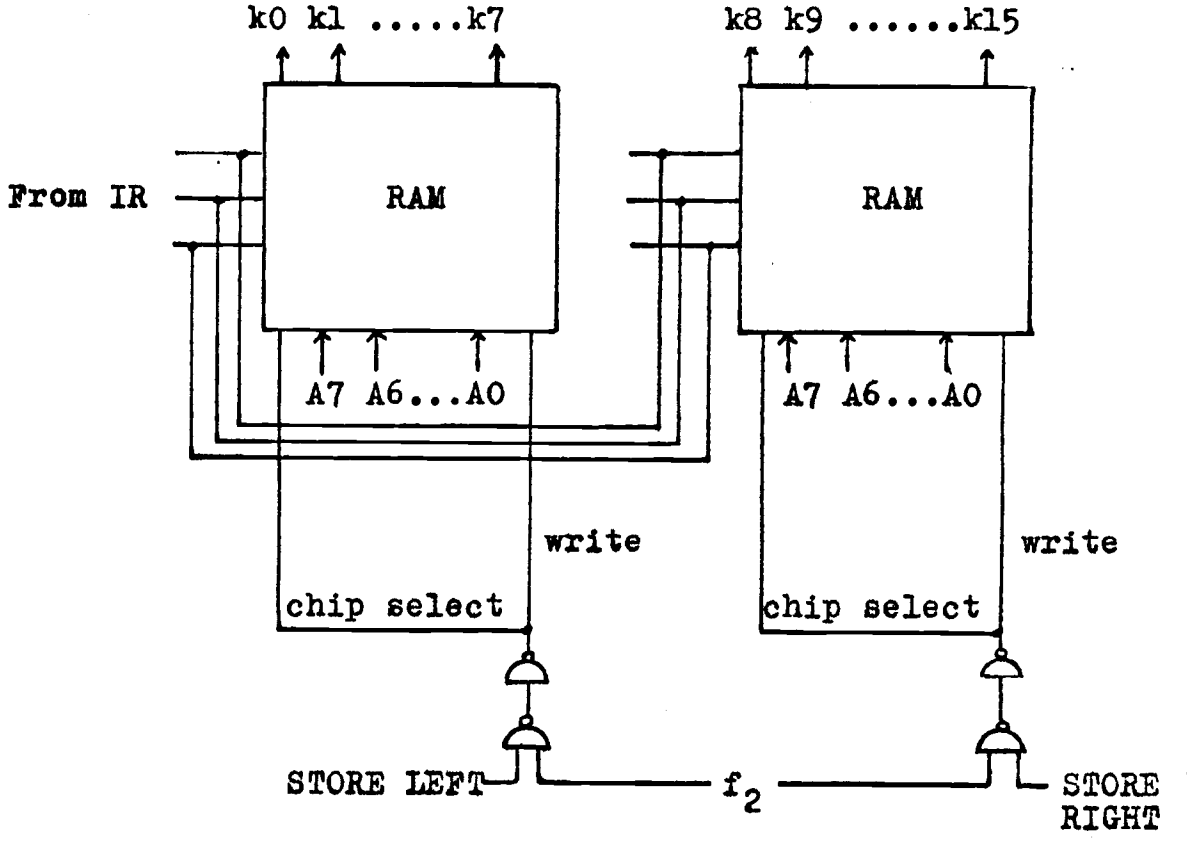
Figure 4.11 (a) Typical Stage of the IR;
(b) The Address Decoder.

back to Table 3.1. If IR7=1 the word location in ROM will be specified by IR7, IR6, and IR5. This is because the op-code of Class(11) commands is also the address of the corresponding function stored in the ROM. If IR7= 0 the address of each function of the Class(01) commands stored in ROM will be specified by IR4, IR3, and IR2.

The ROM and RAM. This computer allows the use of either one of the memories as a function-select matrix. Figure 4.12(a) shows the ROM.



(a) ROM



(b) RAM

Figure 4.12 The Function-Select Matrices.

The STORE LEFT and STORE RIGHT commands are to load the contents of the accumulator into the left and right half of a RAM to constitute a full 16-bits binary function. This can be accomplished by using two 8 x 8 bit RAMs with chip-select* as shown in Figure 4.12(b).

Link. The Link stores overflow from the AU. To be a meaningful overflow the flip-flop is clocked only when commands to be executed belong to Classes (11) and (01). Figure 4.13 is the logic circuitry of the Link register.

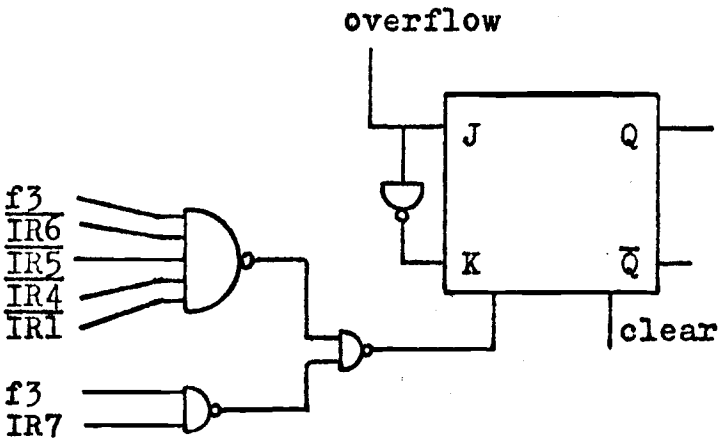


Figure 4.13 The Link.

Accumulator. This register is called upon to store the result of arithmetic operations and the information from external devices to be transfered into the computer.

*Fairchild Semiconductor's MuL 9035.

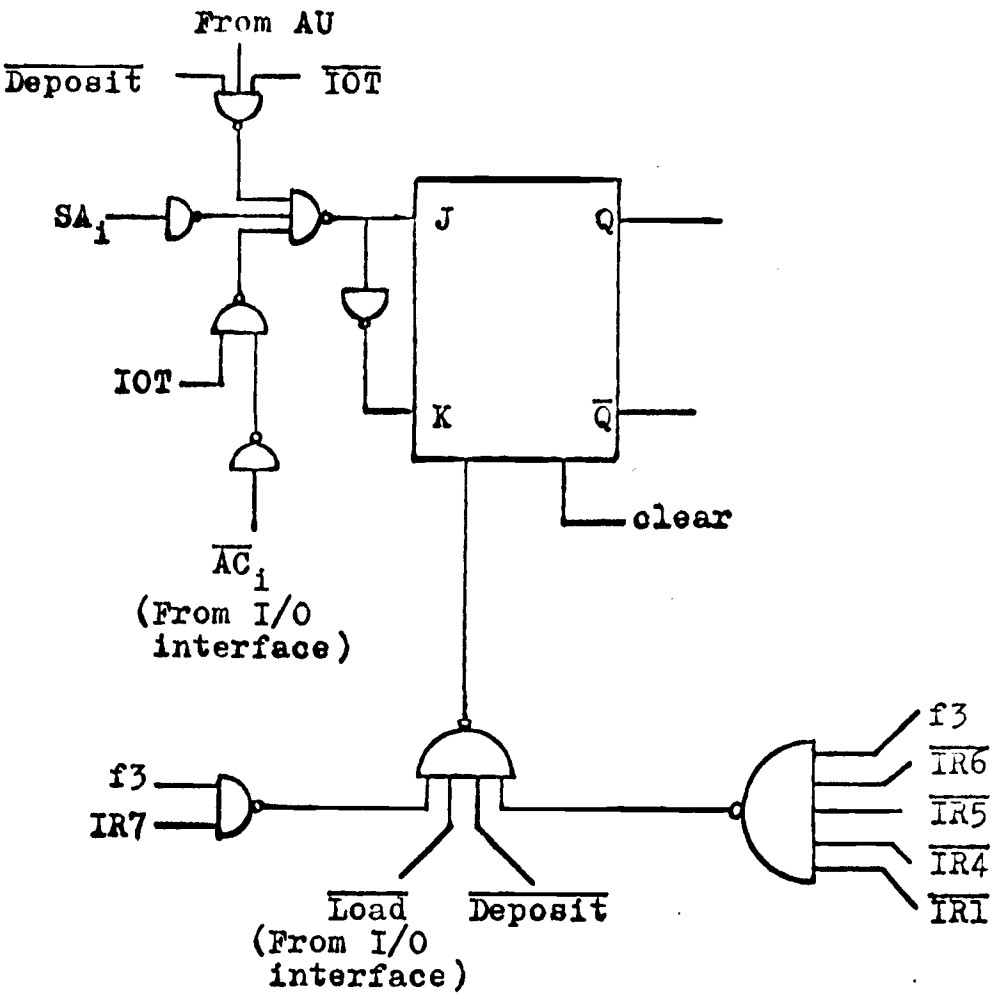


Figure 4.14 Typical Stage of the Accumulator.

Program Counter. From Figure 4.1 of the state diagram, PC is activated when in state f_1, or in state f_2 of the JUMP, BRN, and BRL commands. All of the preceding commands except the JUMP increase the PC by 1. The JUMP loads new contents to the PC. The circuitry is shown in Figure 4.15.

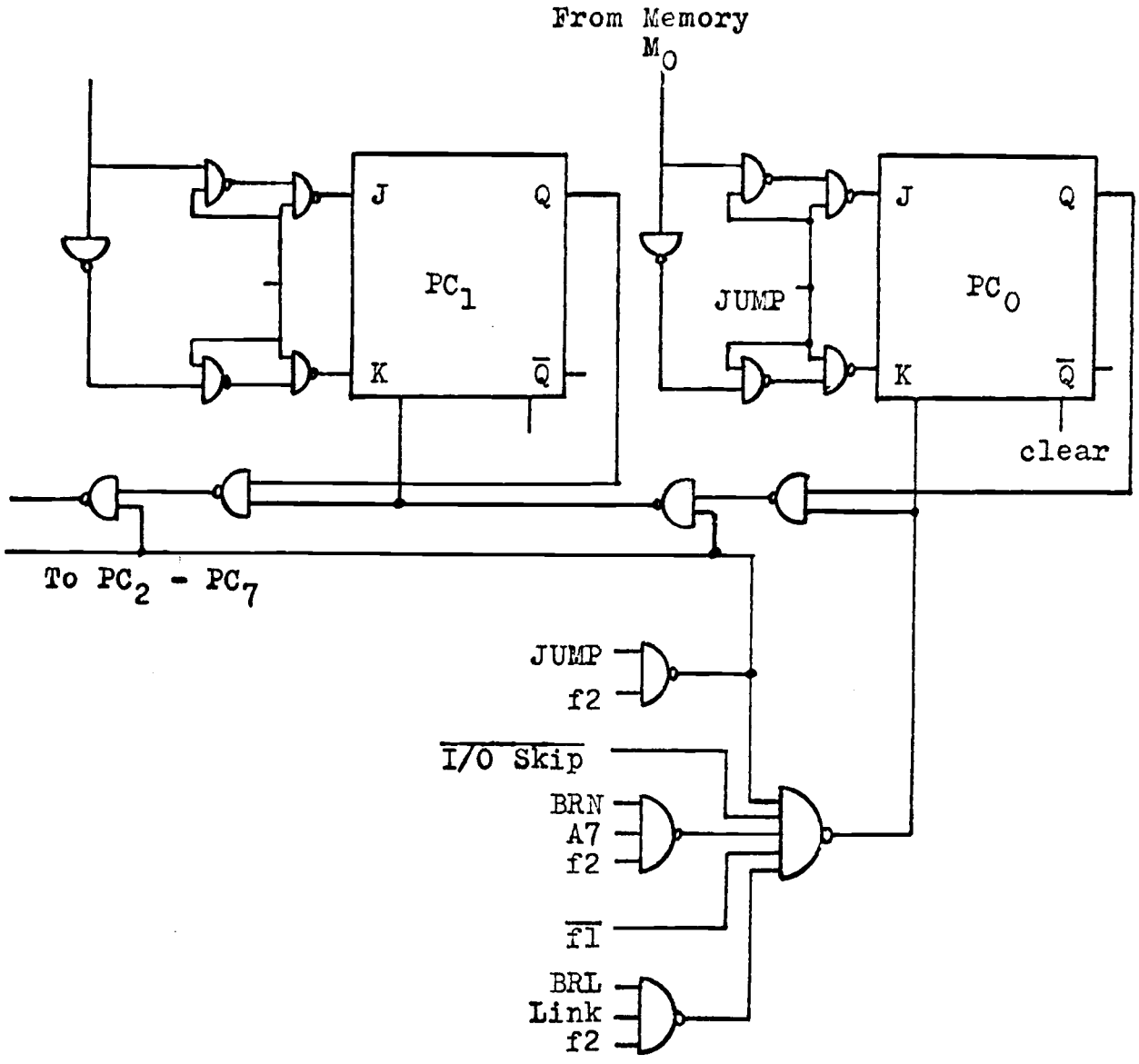


Figure 4.15 The Program Counter.

V. CONCLUSION

It has been demonstrated that this arithmetic unit is capable of performing a large number of operations. Aside from those listed in Table 2.1, the AU could probably realize more instructions since there are a total of 2^{16} functions it can provide. The operating speed of the unit however is slow. A carry generated from the least significant cell has to ripple through each cell before the sum is produced. In other words, the unit cannot have carry-lookahead because of the many varieties of operations involved. But it should also be noted that some instructions performed in one operation by the AU would probably require more operations in another type of machine for the same instructions.

The small digital computer just designed has shown that employing such an AU is feasible. From the economic point of view, the function-select matrix using either ROM or RAM may cost a little more but allows a large instruction set to be selected for a small computer. From the logic circuits design consideration, the large instruction set does not contribute any complexity to the logic networks.

If we refer back to Table 3.1, we will observe that about half of the instructions are performed by the AU. These instructions can be altered without affecting the

hardware design. The other half of the instructions are executed by controlling gates permanently built in the machine. Because of this structure, this small digital computer using ROM in the AU can be considered as another type of machine which is in between the conventional and the microprogramming.

The ideas devised about the AU have been tested out in a machine that was rather simple in organization. For further study, the applicability of these ideas in a more complicated system having double length register, index register, or more data registers could be examined. One might also consider developing the ROM into full micro-programming and investigate the possibility of improving the speed in the arithmetic unit. Traditional speed up operations might also be investigated.

BIBLIOGRAPHY

1. Chu, Yaohan. Digital Computer Design Fundamentals. New York, McGraw-Hill, 1962. 481 p.

2. Digital Equipment Corporation. Small Computer Handbook. Maynard, Massachusetts, 1967.

3. Herzog, James H. Digital System Design. Corvallis, Oregon State University, 1971.

4. Husson, Samir S. Microprogramming Principles and Practices. Englewood Cliffs, Prentice-Hall, 1970. 614 p.

5. Podraza, G.V., Gregg, R.S.Jr., and Slager, J.R. Efficient MSI Partitioning for a Digital Computer. Vol.C-19. IEEE Transactions on Computers, Nov. 1970. p.1020.