SBA

PRELIMINARY INFORMATION

Sequential Boolean Analyzer

FEATURES

- 1023 words of program
- 30 programmable inputs, outputs, or multiplexed input/outputs.
- 16 element stack and 120 element Read/Write memory
- AND, OR, XOR, COMPARE, INVERT basic logic functions
- Serial processing of inputs and stored information provides very easy programming in Boolean logic
- Versatile clock generation scheme
- TTL compatible inputs and outputs
- Simulator and software program compiling facilities available

DESCRIPTION

The SBA is a microprogrammable Sequential Boolean Analyzer which forms the basic controlling element for many systems requiring timing and control functions. The SBA is fabricated in GI's low voltage Ion Implant N-channel process resulting in high speed operation and low power dissipation

APPLICATIONS

The SBA is suitable for a very wide spectrum of applications such as:

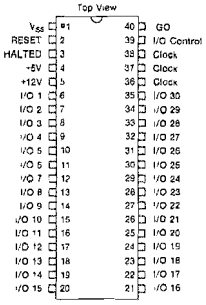
TELECOMS: Simple PAX controllers, Relay circuit control, Answering machine controllers, Line seeker/monitor.

INDUSTRIAL: Complex sequential timers, Small machine controllers, Special purpose digital clocks, Alarm monitor.

CONSUMER: Gaming machines, White goods timers, Combination locks, Pinball machine/one arm bandit.

PIN CONFIGURATION

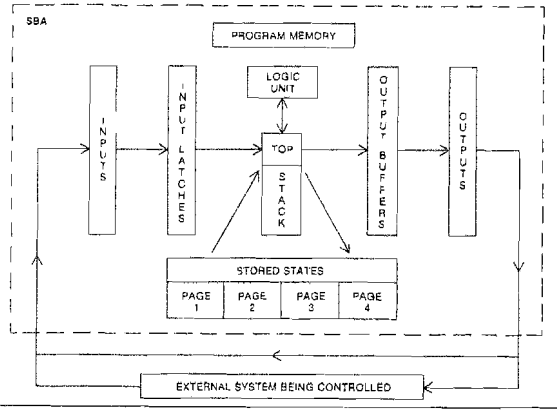
40 LEAD DUAL IN LINE



MICROPROCESSORS: As a slave processor for BCD/binary conversion, Binary/BCD conversion, Alarm condition monitor/interrupt generator, Peripheral controller

In general the **SBA** is best suited to applicetions where a control response is required in milliseconds rather than microseconds

BLOCK DIAGRAM AND DATA PATHS



PIN FUNCTIONS

RESET Input	When taken to a logic '0', this input will reset the program to the start position.
GO Input	When taken to a logic '0', this input will halt the program at the end of the program cycle and will activate the HALTED output When taken to a logic '1', the program will cycle continuously.
HALTED Output	A logic '1' on this output indicates that the program has stopped cycling.
INPUTS/OUTPUTS	There are 30 input/outputs on the circuit which can be mask programmed as inputs, outputs, or inputs/outputs
I/O Control	When this output is at logic '0', the circuit will output information When it is at logic '1', it will read input data.

ELECTRICAL CHARACTERISTICS

Maximum Ratings*

Voltage on any pin with respect to V_{SS} ..... -0.3V to +15V
Storage Temperature Range ..... -65°C to +150°C
Operating Ambient Temperature Range ..... 0°C to +70°C

*Exceeding these ratings could cause permanent damage. Functional operation of this device at these conditions is not implied — operating ranges are specified below.

Standard Conditions (unless otherwise noted)

V_{SS} = 0V
V_{CC} = +12V ± 10%
V_{GG} = -5V ± 15%

Characteristic	Min.	Typ.	Max.	Units	Conditions
Clock	10	—	800	kHz	Note 1
Inputs					
Logic '0' level	+0.3	—	-0.4	V	
Logic '1' level	+2.4	—	+12	V	
Current	—	—	10	µA	V_{IN} = +12V
Timing	—	—	—	—	Note 2
Outputs					
Logic '0'	—	100	—	Ohms	0.7V at 7mA
Sink Current	—	7	20	mA	Max total power 150mW
Logic '1'	1.2	—	—	MOhms	open drain
Leakage current	—	—	10	µA	V_{OUT} = +12V
Timing	—	—	—	—	Note 2

NOTES:
1. Clock frequency controlled by external R/C network
2. The timing of inputs depends on the clock frequency and the program length Refer to the detailed descriptions.

9

PART I General Information

A. INTRODUCTION

The Sequential Boolean Analyzer (SBA) is a very simple single bit processor which can directly evaluate a set of Boolean equations.

The use of Boolean Equetions as a 'programming language' has a number of unique advantages:

1. it is concise
2. it is standarized worldwide
3. engineers already use it and *understand it*
4. universities teach it now and have dona so for many years.
5. it serves the dual purpose of both program and documentation
6. it has stood the test of time.

The equations define the logic that controls the system to which the SBA inputs and outputs are connected.

In eddition to Boolean logic, most systems require that some events have to be 'remembered'. this being the reason for the use of flip flops in TTL type logic implementations. In the SBA a number of internal storage elements are provided for such purposes

A memory is used to hold an encoded version of the Boolean equations that define the desired function of the SBA and there is a one to one correspondence between the data in this memory and the Boolean Equations as written by an engineer.

B. PRINCIPLE OF OPERATION

A block diagram of the SBA showing the program memory, inputs, logic unit, stack. stored states and outputs is shown in Fig. 1.

The SBA functions as follows:

1. The new inputs are read in from the system being controlled and are latched into the input latches.
2. The SBA now reads the Boolean equations out of the memory and, using the logic unit and the stack, it processes the input data and stored states one Boolean term at a time to produce results which are either 'remembered' as stored states or placed in the output buffer.
3. The Boolean equations are taken from the memory term by term and when all the equations have been evaluated the results are transferred from the output buffers to the outputs and thus to the system being controlled. The program address counter is then reset and the cycle begins again.

C. COMBINATIONAL LOGIC

If the SBA is used to emulate combinational logic, then the Boolean equation which defines the logical function will only contain input terms and output terms. For example:

A = B.C.~D+E.F. (X + ~X.G)
L = T + E.F. (X + ~X.G)

where . signifies logical AND
+ signifies logical OR
~ signifies negate (~D read not D)
A and L are outputs
all other letters are inputs.

The number of program steps required to evaluate the above Boolean equations can be reduced by using a stored state to save the value of E.F (X + ~X.G) after it has been evaluated the first time. This partial result can then be used in the second equations. Now we have:

a. stored state = E.F. (X + ~X.G)
b. A = B.C + stored state
c L = T + stored state

This example serves to illustrate how the equations can sometimes be optimized by trading off stored state memory against program memory.

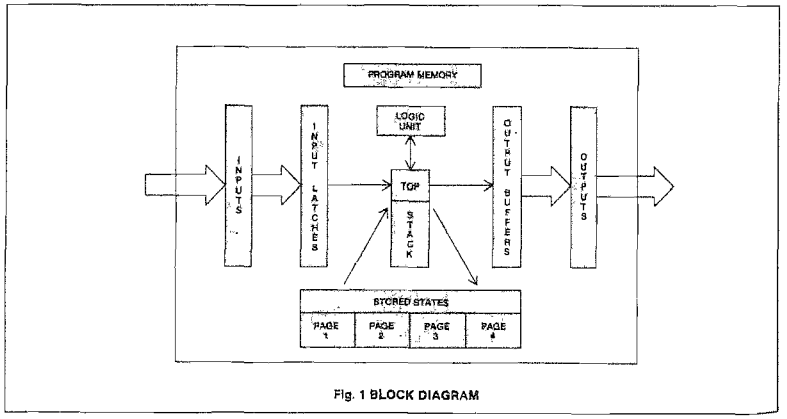


Fig. 1 BLOCK DIAGRAM

D. SEQUENTIAL LOGIC

Although some problems are combinational in nature, the vast majority of practical problems are sequential

A sequential system is one in which the response to a given set of input conditions is dependent on the previous history of the system. An example might be a digital clock where normally the seconds digits are incremented except when they are at 59 when they are reset to 00. In other words the next response of the counter depends on its current value and there may be different responses depending on different current values

All sequantial systems can be described by a combinational network in which some of the results of the Boolean equations are stored in a memory. It is this memory that remembers the history or 'state' of the sequential system.

Thus the stored state memory of the SBA has its main use in remembering the 'state' of the system being implemented. Each time the SBA evaluates the complete set of Boolean equations describing a saquential system, it uses the stored states as part of the equations. As the evaluation proceeds, the stored states may be changed if the Boolean equations demand it

A simple example is shown by a Vending Machine where there are two major states — 1. not enough money to buy anything
2. enough money, so supply the goods

In this example some of the stored states would be used to keep count of the money that has been fed into the machine The Boolean equations controlling the dispensing of the goods would all contain a term involving the stored state that could never be logically true if there were not enough money to buy the goods. As soon as goods were bought and supplied the stored states holding the 'amount' of money would be altered to reflect that the goods had been supplied thus switching the system back to state 1.

E. SYSTEM DESCRIPTION

The Sequential Boolean Analyzer consists of the following major components (refer to Fig 1)

1. A Program Memory which holds the set of Boolean equations defining the system operations. In the single chip version of the SBA this is contained in an on-chip mask programmable ROM. There will be another version in which the program memory is off the chip and it can then be ROM. RAM or PROM as required. The Boolean equations which define the logical relationships between the SBA inputs, stored states, and outputs are stored in the memory as 8 bit words in an encoded form. An exact definition of the code is given in the next section
2. A set of up to 30 input buffers which are latched at the start of the evaluation of Boolean equations. This is done so the input values are consistent during the whole period of time it takes to evaluate all the Boolean equations once
3. A number of pages of 30 stored-state flip flops which can be grouped to emulate counters and shift registers, or used singly as 'flags' to remember the state of the machine. or in logic equation reduction The SBA addressing structure only allows for 30 addresses, out the number of stored states is increased to 120 by having typically 4 pages of 30 each Two instructions control a page counter'. One steps the counter and so changes the page, and the other sets it back to the first or 'home page' Thus an infinite number of pages is theoretically possible, the SBA stepping through them in sequence as required with the option to return to the start at any time
4. A logic unit which can perform all the possible logic functions of two variables, namely AND, OR, EXCLUSIVE OR and COMPARE and also negate (invert) The truth tables of the functions are shown in Fig. 2. Any logic system can be described by a set of Boolean equations written with these operators

   The logic unit always has two inputs and produces one output and there are two types of action.

   (a) One input comes from an input latch or stored state; the second input comes from the top of the stack, and the result is placed on the top of the stack.
   (b) One input comes from the top of the stack; the second input comes from the next location of the stack; and the result is left on the top of the stack

   The exact operation of the various Boolean equation evaluation codes is defined in the next section.
5. The stack is always involved in logical evaluations, as the top of the stack is always one of the operands to the logic unit. The stack is just a pile of Boolean values and can be imagined as a vertical shift register in which data is always put into or taken from the top When data is added to the stack, it is said to be 'pushed' onto the stack The new data becomes the top of the stack and all the previous data is pushed down the stack by one place. If the opposite is performed, the data in the stack is moved up by one place and the stack is said to be 'popped' The top of the stack will be lost and the data previously just below the top will become the new top of the stack.

   The data stack in the SBA is normally used as workspace or accumulator and the top of the stack is used in most of the instructions together, sometimes, with the next location down. However, the stack can also be used to store temporary Boolean variables and helps greatly in the evaluation of Boolean equations containing brackets. For example a function such as

   A. (B.C. + (D.E + F.G).(H.I + J.K))

   would be evaluated in the following way:

Operation	Result
(a) evaluate D.E	D.E
(b) push into the stack	
(c) evaluate F G	
(d) OR	D.E+F.G
(e) push into the stack	
(f) evaluate H.I + J.K as in (a)-(d)	
(g) AND	
(h) push into stack	(D.E+F G).(H.I+J.K)

AND			OR			EXCLUSIVE OR			COMPARE		
A	B	RESULT	A	B	RESULT	A	B	RESULT	A	B	RESULT
0	0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	1	1	0	1	1	0	1	0
1	0	0	1	0	1	1	0	1	1	0	0
1	1	1	1	1	1	1	1	0	1	1	1
Output true only if both inputs true.			Output true if either input true.			Output true when inputs differ			Output true when inputs comoare		

Fig. 2 LOGIC FUNCTIONS

Operation	Result
(i) evaluate B.C	
(j) OR	B.C+(D.E+F.G).(H.I+J.K)
(k) AND the stack with A	Final result.

Naturally this is not the only way the stack can be used, the variation being limited only by the imagination

6. A set of up to 30 output buffers which have their new values stored in them as the Boolean equations are evaluated. When all the equations are complete, at the end of a program cycle, the data on the output buffers is relatched into the output drivers connected to the system being controlled. In this way the external outputs of the SBA are updated once per cycle and remain unchanged until the end of the next complete evaluation of the Boolean equations.

F. INPUT/OUTPUT

It has been noted that there can be up to 30 inputs and up to 30 outputs available in an SBA. However, because of the physical limitation of an actual device, if there are more than a *total* of 30 inputs and outputs then inputs and outputs are multiplexed onto the same pins. So the device will be available with 30 pins mask programmable to be inputs, outputs or multiplexed input/outputs

G. CONTROL LOGIC

As well as the major functional blocks described above, the SBA contains some simple control logic which operates transparently to the user. At the end of a complete program cycle, i.e. at the Restart instruction, the following actions are performed:

(a) the contents of the output buffers are relatched into the outputs
(b) the top of the stack is set to a logic 1
(c) the page counter is set to the home page
(d) new inputs are latched
(e) the program address counter is set to point at the first term of the first Boolean equation.

Once the SBA has started a program cycle, the program address counter is simply incremented every SBA clock cycle, the instruction read out and acted upon.

H. SBA RESPONSE TIME

It is implicit in the description of the operation that the speed of response of the SBA to an external system being controlled is determined by the length of time it takes to evaluate the complete set of Boolean equations once. This is because the inputs and outputs are only latched once per program cycle of the SBA's operation.

The SBA is designed in such a way that all its logical operations and data transfers take the same time to execute (in fact, one of the SBA's internal clock cycles) and so the response time of the output of the SBA to new inputs from the system being controlled by the SBA is directly proportional to the number of Boolean operations required to define the control function. This response time will typically be of the order of a few milliseconds.

I. DATA PATHS

The data paths available in the SBA are illustrated in Fig. 3. The focal point as far as data is concerned is the top of the stack since all data transfer go to or come from the top of the stack. The stack is loaded from an input or stored state. The logic unit can perform any logical function on it and the result can then be stored on any output buffer or a stored state.

Note that it is not possible for Boolean equations to use terms involving data on the output buffers. If such a facility is absolutely necessary, then a copy of the output buffer state must be made in a stored state so that the data path to the top of the stack is made available. The outputs of the SBA can be connected back to inputs either directly or via some piece of circuitry. A direct connection forms a stored state that can be accessed from outside the SBA. This can also be a limited source of extra stored states. External logic can be connected between outputs and inputs or even keys and switches. This latter possibility is useful in, say, scanning a matrix switch or selecting a code switch, and reduces external circuitry.

However, in general, the outputs of the SBA are connected to the parts of the external system that controls its actions and the inputs of the SBA are connected to parts of the system that monitor its current state.

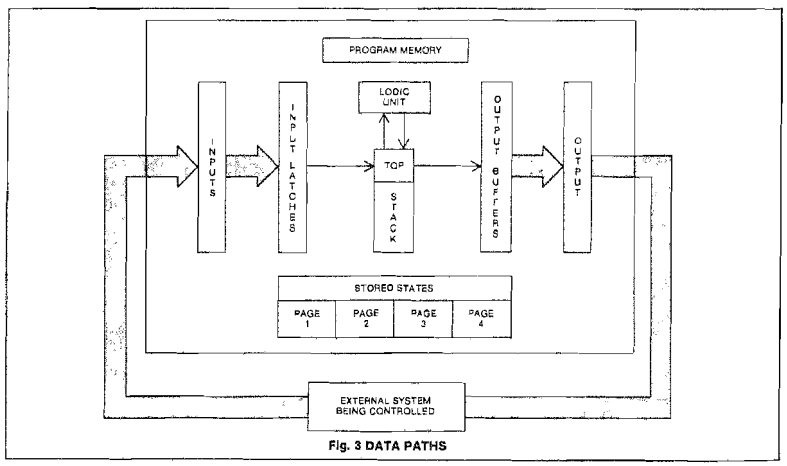


Fig. 3 DATA PATHS

J. BOOLEAN EQUATIONS IN THE PROGRAM MEMORY

In order to make the best use of the space available for memory, the codes representing the Boolean operations should be as efficient as possible. It has been determined that about 20 instructions would provide a good compromize between the number and efficiency of the instructions. Given also that about 30 inputs and outputs were suitable for the requirements of the type of system likely to be controlled by an SBA, the following scheme is used.

An 8 bit word is used for each instruction code. 5 bits provide an address for the inputs, outputs or stored states and, if two of the 32 available addresses are reserved for addressless instructions, the remaining 3 bits of the code enables a total of 24 instructions to be made available.

The 8 bit binary word is conveniently represented by 3 octal (radix 8) digits ranging from 0 to 377 (00 000 000 to 11 111 111 binary). The least significant 3 binary digits are treated as the instruction, and the 5 most significant bits as the address. Addresses 1 to 36 (octal) represent the 30 addresses required throughout the SBA (1-30 in decimal). Addresses 0 and 37 (all 0's and all 1's in binary) are reserved for instructions as shown in Fig. 4.

Memory Code in OCTAL		MNEMONIC	Functional Description of Code
01 to 36	0	ANDIN	This input specified by the address is ANDed with the top of the stack.
01 to 36	1	NANDIN	The logical inverse of the input specified by the address is ANDed with the top of the stack
01 to 36	2	ANDSS	The stored state specified by the address* is ANDed with the top of the stack
01 to 36	3	NANDSS	The logical inverse of the stored state specified by the address* is ANDed with the top of the stack.
01 to 36	4	ASPI	The stored state specified by the address* is ANDed with the top of the stack then a 1 is pushed onto stack
01 to 36	6	NASPI	The inverse of the stored state specified by the address* is ANDed with the top of the stack then a 1 is pushed onto stack.
01 to 36	5	STORE	The value on top of the stack is stored in the stored state specified by the address* and a logical 1 left on the stack
01 to 36	7	OUTPUT	The value on top of the stack is stored in the output buffer specified by the address and a 1 left on the stack.

Memory Code in OCTAL	MNEMONIC	Functional Description of Code
000	RESTART	Restart evaluation of equations
001	INVERT	Invert top of stack
002	PAGE	Change Page
003	HOME	Back to Home Page
004	PUSH 0	Push logic 0 onto stack
005	PUSH 1	Push logic 1 onto stack
006	PUSH C	Push and copy top of stack
007	POP	Pop the stack
370	AND	Perform the function on the top two locations of the stack. Store result on top of stack
371	OR	
372	EXOR	
373	COMP	
374	PAND	Perform the function on the top two locations of the stack Push result into stack leaving logic 1 on top
375	POR	
376	PEXOR	
377	PCOMP	

*The address of a stored state defines 1 of 30 on the currently enabled page.

Fig. 4 BOOLEAN EQUATION CODES

Description of Codes.

The codes used for defining the Boolean equations fall into four categories:

(a) single operand instructions which affect the stack
(b) logical operations taking inputs from the stack and storing the result on the stack
(c) data transfer instructions for inputs, stored states, and outputs
(d) program control instructions.

Category (a) — stack manipulation (See Fig. 5):

1. Invert top of stack. The Boolean value is taken from the top of the stack and replaced by its logical inverse; a 1 becomes a 0, and a 0 becomes a 1. The stack is neither pushed or popped.
2. Push 0 onto stack. All data values on the stack are pushed down one place and a logic 0 entered on top of the stack
3. Push 1 onto stack. All items in the stack are pushed down one place and a logic 1 put on the top.
4. Push and Copy top of stack. Data is moved down the stack and the previous top of stack (now the next position down) is copied to the top
5. Pop the stack. All data values are moved up the stack one place. The old top of stack is lost.

Category (b) — logical operations (See Fig. 6):

There are two types here, a logical function (AND, OR, EXOR, COMP) and the function followed by PUSH:

1 AND, OR EXOR, COMP. The top two elements of the stack are propped into the logic unit; the logical function is performed on them, and the result pushed back onto the stack.
2 AND-PUSH, etc. The top two values on the stack are popped into the logic unit, the appropriate function is performed on them, the result is pushed onto the stack, and this is followed by a logical 1. This form of logical operation is used when the result is to be saved in the stack for subsequent processing. The 1 is put onto the stack to make it ready for further evaluations

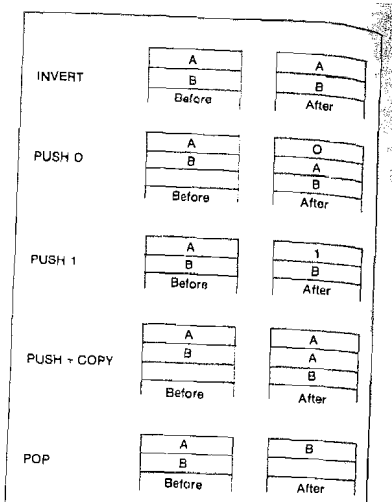


Fig. 5 STACK MANIPULATION

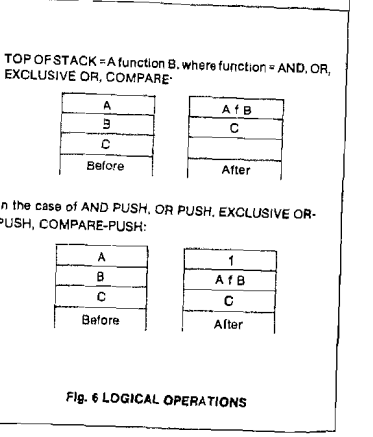
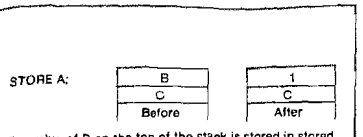
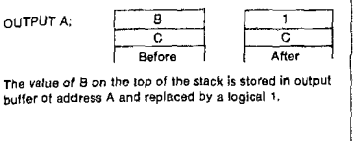


Fig. 6 LOGICAL OPERATIONS



The value of B on the top of the stack is stored in stored state of address A and replaced by a logical 1.



The value of B on the top of the stack is stored in output buffer of address A and replaced by a logical 1.

Fig. 7 OUTPUT OPERATIONS

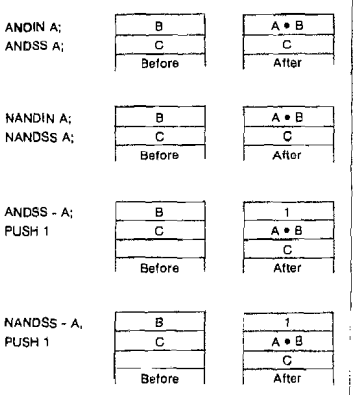


Fig. 8 INPUT OPERTIONS

Category (c) — Input/output operations (See Figs. 7 and 8):

These instructions have two parts, the command and the address. The command defines the data path (input-stack, stored state-stack, stack-stored state or stack-output) and the logical operation to be performed, if any. The address defines which of the 30 pieces of date is to be manipulated

STORE and OUTPUT take the logic value from the top of the stack and transfer it to the appropriate stored state or output buffer, the top of the stack being replaced by a logical 1 ready for the next evaluation.

ANDIN takes the value of the addressed input and ANDs it with the top of the stack.

NANDIN takes the logical inverse of the addressed inputs and ANDs it with the top of the stack.

ANDSS takes the value of the addressed stored state and ANDs it with the top of the stack.

NANDSS takes the logical inverse of the addressed stored state and ANDs it with the top of the stack.

In the above four operations, the stack is neither pushed nor popped.

ASP1 takes the value of the addressed stored state, ANDs it with the top of the stack, and pushes 1 onto the stack

NASP1 takes the logical inverse of the addressed stored state, ANDs it with the top of the stack, and pushes 1 onto the stack

Category (d) — control:

PAGE There are typically 4 pages of stored states in the SBA, and the instructions reading and writing to and from the stored states only provide an address within the currently enabled page. The PAGE instruction steps the page counter and enables the next page.

HOME. If it is required to enable a page of stored states that has been passed, the HOME instruction causes the page counter to go back to enable the home page.

As an example, if the page counter is currently enabling page 3 and it is required to update a stored state on page 4, the PAGE instruction would be used. If now a stored state on page 2 is required, a HOME instruction will switch back to page 1 and a PAGE instruction will step to page 2

RESTART. This code is always the last code in the program memory (note that its value is conveniently all zeros) and when seen by the control logic the following is performed:

1 the contents of the output buffers are relatched to the outputs.
2. a new set of inputs are latched
3. the top of the stack is set to logical 1
4. the page counter is reset to the home page
5. the program address counter is reset to restart the evaluation of Boolean equations

PART II Using the SBA

A. INPUT/OUTPUTS

The addressing capability of the SBA allows for 30 inputs and 30 outputs. There are 30 pins available as Input/Outputs and so if more than 30 total inputs and outputs are used they must be multiplexed

There is a mask programmable option on each of the 30 pins to allow them to be inputs, outputs or multiplexed input/outputs. Regardless of the option, the internal addressing of the pins remains the same and so, for example, if an application requres only one input and the board layout requires it to be the last pin the program must use eddress 30.

Input

When programmed as an INPUT,nothing will be able to be output from the pin even if the program loads something in the respective output buffer.

Output

When programmed as an OUTPUT, the input date path is still connected and the value on the pin will be latched with all the other inputs at the appropriate time. This fact can be utilized as follows:

(a) It is sometimes required that the value of an output is used in the processing during the following cycle and would normally have to be copied into a Stored State. Feeding an output back as an input in this way avoids this problem and can also be used as a limited supply of extra stored states, if spare pins are available, that can also be read outside the chip.

(b) As well as reading the value of the output directly, the output can be modified by making use of the open drain construction of the output driver. The value of the output can be modified by an externally connected active pull-down device and the result read into the input. Something like a manual override would be a simple use of this facility while more complex logic functions can also be easily performed.

Multiplexed Input/Output

For complete separation of inputs and outputs the MULTIPLEXED facility must be used although external logic must be used for demultiplexing.

Output Drive Capability

The Output drivers have an impedance of 100 Ohms and are nominally rated at a sink current of 7mA. Thus each output can drive 4 TTL loads plus a 10 KOhm pull-up resistor to +5V.

The nominal current rating is determined by the total allowable power dissipation in the output circuits, which is a maximum of 150mW. The 7mA rating has been determined assuming all 30 outputs are being used. If less than the maximum are in use, the current rating for each can be increased up to the maximum rating of 20mA, keeping within the 150mW power restriction. Fig. 9 gives a guide to the current capacity for different numbers of outputs in use.

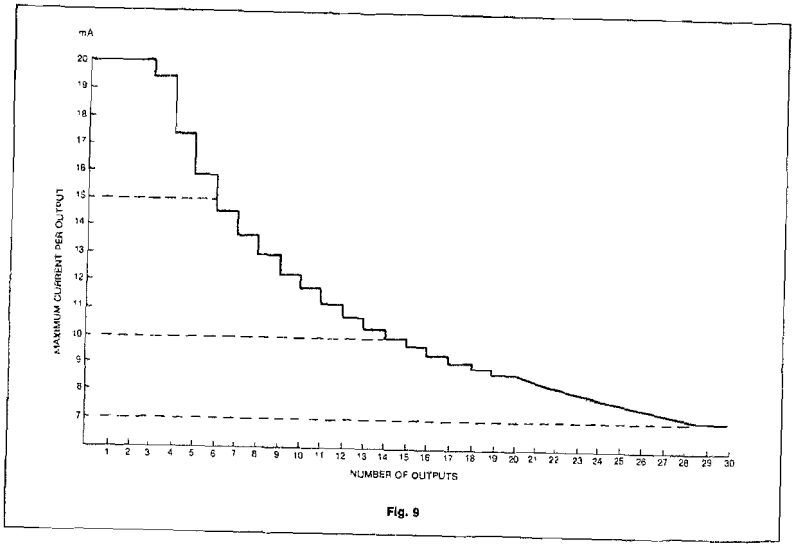


Fig. 9

B. DEVICE TIMING

The clock can be generated internally with the help of an external R/C network connected to the three 'clock' pins, or an external clock can be applied to the clock input.

The SBA will perform the instructions in the ROM, one per clock cycle, until the RESTART instruction is reached. At this point a special sequence is performed that is shown in Fig. 10.

The possible variations are due to separate or multiplexed inputs/outputs and the use of HALTED and GO.

Input Timing

Inputs are stroped into the device once per program cycle, as shown in Fig. 10, whether they are exclusive inputs or multiplexed.

Output Timing

If a pin is dedicated to the output function, then the logic level remains constant throughout the program and is updated at the end of each cycle as shown in Fig. 10.

In a multiplexed I/O the output data is present for two clock cycles at the end of each program cycle, the I/O control providing a strobe so the data can be stored in an external device (on the negative edge).

Modes of Operation

1. If the SBA is required to cycle continuously the GO input will be true in this case there will be one clock cycle between I/O control disappearing and the SPA strobing the inputs This is the more common mode of operation.

2. If GO is not present after the I/O control goes high, the HALTED output will appear and the SBA will stop. When GO becomes true, HALTED will be removed and the SBA will continue by latching a new set of inputs. Although this mode can be useful it is usually better to put the control into the program itself and run continuously.

Cycle Times

Since all instructions have been arranged to take the same time, the total time taken for a complete program cycle is the time for 1 clock cycle multiplied by the total number of instructions in the program (including RESTART) plus 6 (for the end of cycle sequence). If the cycle is stopped by removing GO, the time can be up to 1 clock cycle less. Thus the maximum possible cycle time is:

Clock period x (1023 + 6) = 1.28625ms at 800KHz
= 10.23ms at 100KHz

In general wa have:

$$\frac{\text{no. of instructions} + 6}{\text{clock frequency in KHz}} = \text{program cycle time in ms.}$$

C. BASIC PROGRAMMING

The internal operation of the SBA is of no concern to the user, the device simply being thought of as a variable array of logic. It can be treated as abstract logic or as the logic family most familiar to the engineer, although speed and response time must be considered separately as it is somewhat unique in the SBA

The logic representing the function to be performed by the SBA is described by a set of Boolean Equations. These can be of any length and are composed of the four logical functions, AND, OR, EXCLUSIVE OR and COMPARE, together with invert or negate and as many levels of brackets as are required Rules and restrictions are minimal.

Stored States — Basic Storage

The Stored States can be used to store the state of some input or logical combinations from one program cycle to the next. For example, if it were required to produce an output whenever an input changed state, the value of the input must be stored and compared with the new input in the next cycle. The equations might be.

output = input * store; (where * represents exclusive OR)
store = input:

Thus, when the input is different from the stored value. I.a. the state of the input last time, the output is produced.

It should be noted here that because of the sequential nature of the SBA the equations are performed one at a time. So the 'store' equation must come after the 'output' equation otherwise the

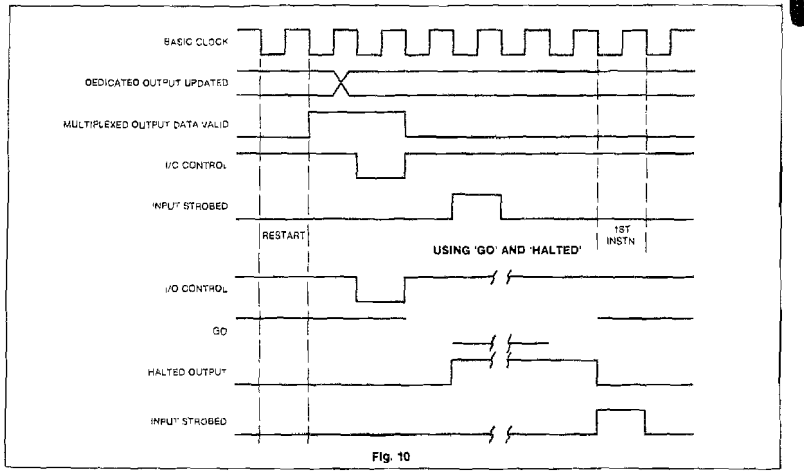


Fig. 10

store would contain the same value as the input of the current program cycle.

The Stored States can also be considered in groups for storing numbers, machine states, etc., and also for counters and sequence generators. Again the equations can either be generated purely logically from a truth table of the required sequence or by considering an appropriate hardwre solution.

Stored State — Temporary Storage

Stored States do not necessarily have to be used as stores from one cycle to the next. It is often convenient to use these as temporary stores within a program cycle. A typical example is when a calculated value is to be used in several equations. Rather than calculate the value each time, it can be calculated once and stored and then read directly from the store as required. Such 'temporary stores' may be updated as many times as required in a program cycle and it is often convenient to reserve a couple of stored states on each page for this purpose. An example might be the Stored State 'temp' being used as a reset for a store and then later as an enable for an output:

```
temp = reset logic;
store A = (store A logic).-temp;
store B = (store B logic).-temp;
etc., then later in the program
temp = enable logic;
output A = (output A logic).temp;
output B = (output B logic).temp;
etc.
```

If the SBA compiler is used, the name for the temporary store must be the same (as 'temp' above) to define the same bit and it can be seen that there might be confusion in knowing the function of the store in a particular part of the program. The problem can be solved by knowing that SBACOMP only takes the first 4 characters of the name. So as long as these remain constant, the same bit of storage will be used and the name can be extended to indicate the current use of that particular store. In the above example the names could have been 'tempreset' and 'tempenable'.

Order of Equations

The most important rule in SBA programming has already been mentioned and that is the order of the equations. While the timing as seen external to the device is clearly defined, it must always be remembered that the equations within a program cycle are performed in sequence and care must be taken to ensure that stores are updated at the right time.

The first example illustrates the overall construction of a program and is demonstrated by considering the store defining the current state of the machine. The state will be output to the system being controlled and the inputs read back from the system. These, together with the present state, will define the next state to be output to the system. When the state number is used in the SBA equations, the position of that equation in relation to the equations updating the state will obviously define which state is being referred to. The state number before updating will refer to the state output to the system last time and will therefore be associated with the current inputs since these are a direct result of that state. The state number after updating will refer to the next state and will be used in determining the outputs.

Another important time to watch the updating of stores is when using such things as counters. A 3 bit binary counter stepped by 'step' might be written:

```
A = A*step;
B = B*(step.A);
C = C*(step.A.B);
```

where the least significant bit changes whenever 'step' is true and the subsequent bits only when all previous bits are true AND 'step' is true. It can be seen that the equations cannot be written as shown because the values on the right hand side refer to the present value of the counter and those on the left hand side the next value. After the first equation, A has been updated and so in the next equation the wrong value will be read. (Note that a hardware solution would update all bits in parallel.) In this case the stack can be utilized to advantage by storing the new values as they are calculated and only when all bits are completed are the stored states updated from the stack. The example would actually be written:

```
stak = A*step;
stak = B*(step.A);
stak = C*(step.A.B);
C = save;
B = save;
A = save;
```

Equations 3 and 4 can be combined but, if written like this for clarity, they will, in fact, be combined by the compiler.

The Pages of Stored States

The order of equations and the variables within the equations can also be important (as far as economy of instructions are concerned) when the different pages of Stored States are used. Equations using variables from more than one page can, in extreme circumstances, use more page change instructions than actual processing!

For this reason, those bits commonly used together (e.g. counters) should be grouped on a single page. It is better to leave a page partly unused in order to do this. Thus four 15 bit counters would certainly be better put one counter per page with the odd associated bits (like resets) kept on their respective pages, rather than squeezing onto two pages and having to put resets etc., on a different page.

If possible, processing should be performed in the order of pages. Thus page 1 processing would be done first, then page 2 and so on to avoid unnecessary page changes. It is easiest to write the program first and then allocate the stores to pages to get the flow right. Data transferring from one page to another can often be stacked first as demonstrated in the following.

Bits on page 1 labelled 1A, 1B, 1C to be moved to page 2, bits labelled 2A, 2B, 2C, and if written:

```
2A = 1A;
2B = 1B,
2C = 1C;
```

will use 11 instructions while,

```
stak = 1A;
stak = 1B;
stak = 1C;
2C = save;
2B = save;
2A = save;
```

uses only 9. If the pages were 1 and 4, the number of instructions would be 17 and 11; if moving page 3 to page 2 we would have 14 against 10. Obsessive page instruction saving is, however, not usually required and is only important when using the ROM to its maximum capacity. It is generally best to write the equations initially using the simple basic rules and to resort to more clever reduction techniques if the ROM is filled.

D. PROGRAMMING EXAMPLES

Combinational Logic

Little needs to be said here since any boolean equations is valid and can include AND, OR, EXOR, COMPARE and INVERT operators and as many variables and brackets as are required.

For greatest economy of instructions the equation should be minimized as far as possible, using EXOR and COMP if appropriate, and in general sums of minterms are most efficient.

If required, even further reductions can be made by considering the instruction set. Inversion of single terms costs nothing, while inversions of multiple terms costs one instruction. The AND operator with a single term is implicit with the reading function, while all other operators and the AND with a multiple term all cost an extra instruction.

A good rule is to use as few OR, EXOR and COMP operators as possible (also AND and INVERT outside brackets). The difference in the number of instructions taken by two versions of the same equations is approximately the difference in the numbers of these 'bad' operators (shown underlined).

Examples:

(a) -A.-B ± -C.-D ± -E.-F instead of (A ± B).(C ± D).(E ± F) saves 3 (2 versus 5)

(b) -(-A.-B.-C.-D) instead of A ± B ± C ± D saves 2

Latches

The straightforward storing, temporarily or otherwise, of input data or logical combinations has been discussed and needs no further comment.

However, it is often required that a piece of data be latched into a store, and in hardware solutions devices such as S-R, J-K and D type latches are available for this function. The stored states in the SBA together with a bit of logic can be made to act as latches, the type being limited only by the imagination. A few examples are as follows:

(a) Simple S-R latch that is set by the variable 'set' and reset by 'reset': Q = set + Q -reset

(b) In (a) the 'set' will override if both inputs appear together. If 'reset' is to override Q = (set + Q) -reset

(c) Clocked latches have greater variety, a simple D type being: Q = data.clock + -clock.Q

(d) A clocked set - reset:
Q = (set.clock) + Q.-(reset.clock)

(e) Clocked set, asynchronous reset, set override
Q = (set.clock) + Q.-reset

(f) Clocked set, asynchronous overriding reset:
Q = ((set.clock) + Q).-reset

(g) Full J-K latches:
Q = J.clock.-Q + Q.-(K.clock)

It is usually best to design the latch to fit the particular requirement since, in general, the more features required the more instructions are needed. If a latch is required to change on a particular edge of a clock, another store is required and we have, for example:

```
clock = inclock.-storeclock;
storeclock = inclock;
```

to detect the 0-1 transition. Either or both edges can be detected in this way. It can be noted here that once an input clock has been monitored for a transition then anything in the whole of that program cycle can be clocked by the same edge.

Counters and Sequence Generators

There are several ways in which counters can be programmed, but a good general purpose method is described here.

The counter will have an input to tell it when to count and let this be called COUNT.

Consider each bit of the counter in turn, from the least to most significant bits, and determine the conditions that require that bit to change. This condition, ANDed with COUNT, can be made to change the bit using the exclusive OR instruction. So for the Rth bit, if the change condition is CR we have:

```
R = R*(COUNT.CR);
```

Care must, of course, be taken to ensure that bits are not used after they have been updated. They can be stored conveniently on the stack. In a binary counter each bit changes when all bits of lower significance are true, and we have, for a 4 bit counter as an example

```
stak = A*COUNT;
stak = B*(COUNT.A);
stak = C*(COUNT.A.B);
stack = D*(COUNT.A.B.C);
D = save,
C = save;
B = save;
A = save,
```

If the counter is to step each program cycle (for a switch scanner or a counter using the cycle for its timing, for example), COUNT can be removed and the first equation changed to stak = -A;

The binary case is very regular and an alternative approach, which is more economical for large (over 4 bits) counters, uses temporary stores to build up the change conditions progressively:

```
T1 = COUNT.A
A = A*COUNT
T2 = T1.B
B = B*T1
T1 = T2.C
C = C*T2
T2 = T1.D
D = D*T1
etc.
```

Note that this time stacking has been avoided since the change condition is built up progressively. It can also be seen that only two temporary stores are required.

The technique is further illustrated by considering the following 'random' sequence:

State	DCBA
1	0000
2	0111
3	1110
4	0001
5	1011
6	0011
7	0100
8	1001
9	0000,

Examining each bit we see that:

```
A changes after states 1, 2, 3, 6, 7, 8
B changes after states 1, 3, 4, 6
C changes after states 1, 3, 6, 7
D changes after states 2, 3, 4, 5, 7, 8
```

Assuming the non-valid states cannot occur, and using standard reduction techniques, the change conditions reduce to:

```
CA = -A + B.-D + -B.D
CB = -C.-D + C.D
CC = -A + B.-C -D
CD = C + D + A -B
```

Now including STEP, and reducing still further for the SBA using exclusive OR(*) and compare (#), we have finally:

```
stak = A * STEP.(-A + (B*D)),
stak = B * STEP (C#D),
stak = C * STEP (-A + B.-C.-D);
stak = D* STEP.(C + D + A.-B),
D = save,
C = save;
B = save.
A = save.
```

And, for completeness, the carry output is given by A.-B.-C.D, reducing to

```
COUT = -B.D;
```

which should go at the beginning, before the states are updated