SLAC-PUB-2205
October 1978
(I)

MICRO-CIRCUITS FOR HIGH ENERGY PHYSICS*

Paul F. Kunz

Stanford Linear Accelerator Center
Stanford University, Stanford, California, U.S.A.

ABSTRACT

Microprogramming is an inherently elegant method for implementing many digital systems. It is a mixture of hardware and software techniques with the logic subsystems controlled by "instructions" stored in a memory. In the past, designing microprogrammed systems was difficult, tedious, and expensive because the available components were capable of only limited number of functions. Today, however, large blocks of microprogrammed systems have been incorporated into a single I.C., thus microprogramming has become a simple, practical method.

1. INTRODUCTION

1.1 BRIEF HISTORY OF MICROCIRCUITS

The first question which arises when one talks about microcircuits is: What is a microcircuit? The answer is simple: a complete circuit within a single integrated-circuit (I.C.) package or chip. The next question one might ask is: What circuits are available? The answer to this question is also simple: it depends. It depends on the economics of the circuit for the semiconductor manufacturer, which depends on the technology he uses, which in turn changes as a function of time. Thus to understand what microcircuits are available today and what makes them different from those of yesterday it is interesting to look into the economics of producing microcircuits.

The basic element in a logic circuit is a gate, which is a circuit with a number of inputs and one output and it performs a basic logical function such as AND, OR, or NOT. Figure 1 shows the basic gate used in the popular TTL technology. It performs the NAND function, that is only when both inputs are TRUE does the output become FALSE. The truth table which describes the operation of the gate would then look like that shown in figure 2 . From this basic gate one can form other logical functions. For example, the NOT function can be generated by tying the two inputs together as shown on the left of figure 3 . It is usually represented by the INVERTER symbol as shown on the right of the figure. Another example is the OR function which may be generated from the NAND gates as shown on the left of figure 4 and usually represented by the symbol shown on the right of this figure. It can be shown that all the Boolean operations can be generated with combinations of the basic NAND gate.

The cost of a integrated circuit depends on the number of gates required to perform the desired function, but the cost of a gate depends on the number of gates in the chip. Figure 5 is a plot of

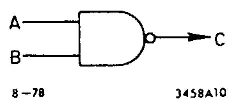


Figure 1: Basic TTL Gate

A input	B input	C output
false	false	true
false	true	true
true	false	true
true	true	false

Figure 2: Truth Table for NAND Gate.

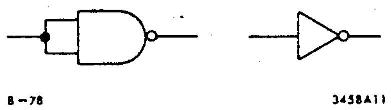


Figure 3: Logical NOT Circuit.

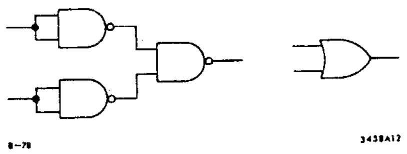


Figure 4: Logical OR Circuit.

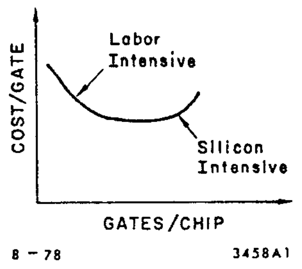


Figure 5: Integrated Circuit Cost Curve.

* Work supported by U. S. Department of Energy, EY-76-C-03-0515

the cost per gate versus the number of gates per chip. There are three distinct regions on this curve. The labor intensive region is where the labor of assembly, testing, and processing the order, as well as the fixed company overhead dominate the costs of the chip. In this region the manufacturer can double the number of gates on the circuit without changing its cost, thus the cost per gate would drop a factor of two. The silicon intensive region is the technically difficult region, where the manufacturer produces a small percentage of functioning circuits for his effort and hence the cost per circuit begins to rise rapidly. The flat central region is the region, where the cost of the circuit is proportional to the number of gates on the circuit. It is the optimal region for producing circuits.

As the technology of producing circuits improved, what was technically difficult at one time became standard practice at a later time. Figure 6 shows the cost curve for three periods of time. These periods correspond roughly to three generations of microcircuit manufacturing. The optimal region in the first generation, Small Scale Integration (SSI), had three to six gates per circuit. The circuits that were produced were simple logic functions and the technically difficult was a flip-flop. An example of an SSI integrated circuit package is the 7400 as shown in figure 7. It is simply four independent NAND gates requiring 3 pins each. With the supply voltage and ground pins it makes the standard 14 pin package still in use today.

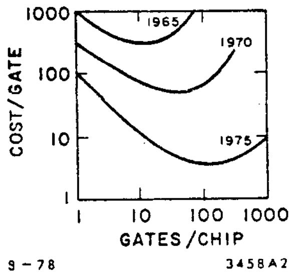


Figure 6: I.C. Cost Curve versus Time.

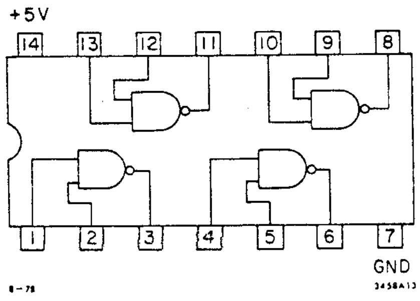


Figure 7: 7400 Integrated Circuit Package

When the optimal region for manufacture became 20 to 50 gates per circuit, the second generation of microcircuits was born: Medium Scale Integration (MSI). The semiconductor manufacturers faced a problem as to what circuits to produce, since the simple extrapolation of more simple logic functions per circuit runs into some problems such as too many pins per package. The problem was solved by producing larger blocks of digital systems such as counters, multiplexers, decoders, registers, etc., which were of general enough use that the manufacturer could sell them in large enough quantities to make a profit. An example of an MSI integrated circuit package is the 74157 as shown in figure 8 (a). It is called a Quad 2-Input Multiplexer since it multiplexs one of two inputs to one output four times over. A single Select input controls all four channels. With one pin left over to make it an even number, the manufacturers have added a Gate to force the outputs to Zero and one has a standard 16 pin package. The conventional symbol for this circuit is also shown in figure 8 (b).

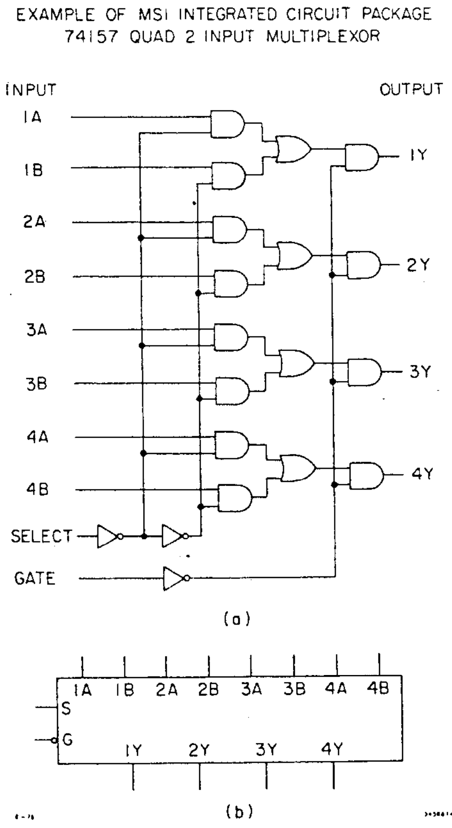


Figure 8: Example of MSI Integrated Circuit Package, (a) Circuit, (b) Symbol.

A few years ago, the optimal region of manufacture became 200 to 500 gates per circuit, Large Scale Integration (LSI), and the semiconductor manufacturers were again faced with the problem of what circuits to provide with these many gates. The problem was solved by producing an even larger block of digital systems so that we now find that microcircuits are arithmetic/logical processor elements, microprogram sequencers, direct memory access controllers, etc.

The LSI microcircuits will be the topic of these lectures. They offer the best economy because large subsystems of digital circuits are available on a single I.C. package. Within a given type of technology (e.g. TTL, ECL, MOS, etc.) they often produce faster systems because there is less lost of speed with interconnection between packages. They also reduce the amount of circuit board real estate required for a given logic system and large systems are less expensive to make.

With LSI microcircuits, the semiconductor manufacturers have made available large digital subsystems within a single I.C. But they still had to provide a means by which the circuit was flexible in its use in order to be able to sell enough of them to make a profit. The flexibility of these circuits was obtained in part by designing them to be used in a microprogrammed type of architecture. That is to say, that the function a circuit performs is controlled by an number of input signals which form an instruction word. The instruction word is assumed to come from the microprogram memory. The manufacturer also is making circuits for use where there is potentially the largest volume of users, which for digital systems is probably the computer and computer peripheral manufacturers. In this market, the microprogrammed technique of logic design offers many advantages as we will see in these lectures.

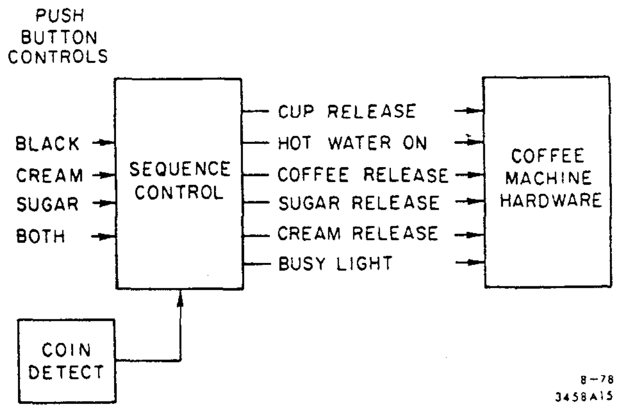


Figure 9: Block Diagram of Coffee Vending Machine

STEP NUMBER	CUP RELEASE	WATER ON	COFFEE RELEASE	SUGAR RELEASE	CREAM RELEASE	BUSY LIGHT	COMMENTS
1	X					X	START
2						X	X = ALL SEQUENCES
3						X	S = SUGAR SEQUENCES
4						X	C = CREAM SEQUENCES
5		X				X	
6		X				X	
7		X	X			X	
8		X	X			X	
9		X	X			X	
10		X	X			X	
11		X	X			X	
12		X		S		X	
13		X		S		X	
14		X		S		X	
15		X			C	X	
16		X			C	X	
17		X			C	X	
18		X			C	X	
19		X				X	
20		X				X	
21		X				X	
22		X				X	
23		X				X	
24		X				X	STOP

Figure 10: Coffee Machine Combined Sequence List.

High Energy Physics is not a high volume user for semiconductor manufacturers. If we are to make use of LSI, we must, in general, bend our needs to those circuits which are already commercially available. In addition, in order to profit from the LSI microcircuits, we must learn the microprogram method of implementing digital systems, and we must be able to understand the digital subsystems that are available as a single I.C. In the following sections, we will first study the basics of microprogramming from a point of view which is biassed by the microcircuits that are commercially available. Then we will study in some detail a microprogrammed controller with a High Energy Physics application. Finally we will study two of the most important LSI circuits which have become available.

2. BASICS OF MICROPROGRAMMING

2.1 COFFEE VENDING MACHINE

To understand the basics of microprogramming let us take a simple example: an automatic coffee vending machine. Figure 9 is a block diagram of such a machine which has two basic parts; the machine hardware and the sequential control logic. The coffee machine hardware is the system to be controlled. It contains the values and solenoids that release the water, coffee, sugar, etc. which are needed to produce the desired result: a cup of coffee. The sequential control logic is the system controller. It sends signals to the hardware in the correct order and timing. It starts the hardware into operation when it receives a signal from the coin detection logic and alters the sequence according to what kind of coffee has been requested via the front panel push buttons.

The sequence control can be imagined as a series of steps, each lasting a fixed length of time, say 1/2 second. The list of steps might be as shown in figure 10 . The coffee machine sequence controller could be implemented using combinations of flip-flops and one-shots as shown in figure 11 . This approach is commonly called hard wired or random logic, and is typical of how designs have been done in the past. The advantage of this approach is that it uses the minimum number of logic gates and it is relatively simple for a given sequence.

The coffee machine sequence control may also be implemented with a binary counter and a read only memory (ROM) as shown in figure 12 . In this figure, only one of the sequences has been implemented. The binary counter serves to count the steps and the ROM serves as a programmable decoder to produce the required signals at each step. Note that the input address of the memory is the output of the counter and that each bit of the memory's output is used directly as one of the signals for the hardware under control. In order to do the black coffee sequence, one would want the contents of the memory to be as shown in figure 13 . A binary '1' corresponds to sending a signal, while a binary 'Ø' corresponds to not sending a signal. A coffee machine sequence controller implemented in this way is said to be microprogrammed.

In order to include the other kinds of coffee one could increase the size of the counter from 5 bits to 7 bits and the size of the memory from 32 locations to 128 locations as shown in figure 14 . The encoder circuit generates a binary code from Ø to 3 depending

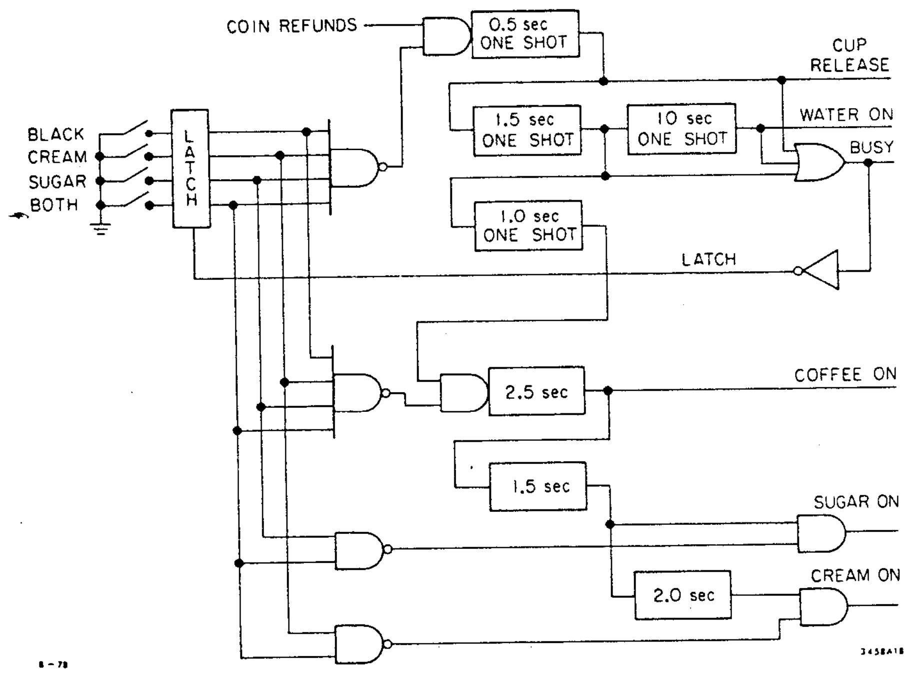


Figure 11: Random Logic Implementation of Coffee Vending Machine.

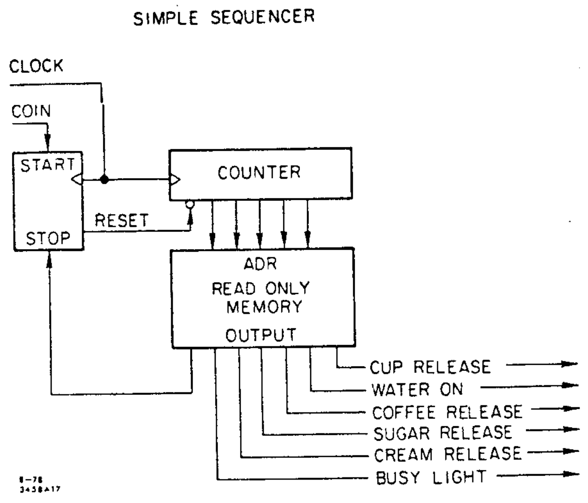


Figure 12: Microprogrammed Coffee Vending Machine.

on which of the push buttons was activated. This code is then used as the two high order bits to the counter when it is loaded. The loading of the counter is under control by one additional bit of output from the memory. Thus, for example, at memory address 1 the load bit may be turned on so that the next address of the sequence will be either 2, 34, 66, or 98 depending on the output of the encoder.

	CUP RELEASE	WATER ON	COFFEE RELEASE	SUGAR RELEASE	CREAM RELEASE	BUSY LIGHT	STOP	
PROGRAM ADDRESS	BIT NUMBER 0	1	2	3	4	5	6	
0	0	0	0	0	0	0	0	IDLE
1	1	0	0	0	0	1	0	START
2	0	0	0	0	0	1	0	'1' = SIGNAL ON
3	0	0	0	0	0	1	0	'0' = SIGNAL OFF
4	0	0	0	0	0	1	0	'X' = DON'T CARE
5	0	1	0	0	0	1	0	
6	0	1	0	0	0	1	0	
7	0	1	1	0	0	1	0	
8	0	1	1	0	0	1	0	
9	0	1	1	0	0	1	0	
10	0	1	1	0	0	1	0	
11	0	1	1	0	0	1	0	
12	0	1	0	0	0	1	0	
13	0	1	0	0	0	1	0	
14	0	1	0	0	0	1	0	
15	0	1	0	0	0	1	0	
16	0	1	0	0	0	1	0	
17	0	1	0	0	0	1	0	
18	0	1	0	0	0	1	0	
19	0	1	0	0	0	1	0	
20	0	1	0	0	0	1	0	
21	0	1	0	0	0	1	0	
22	0	1	0	0	0	1	0	
23	0	1	0	0	0	1	0	
24	0	1	0	0	0	1	1	STOP
25			·	·	·			LOCATIONS 25-31
31			·	·	·			ARE NOT USED

Figure 13: Memory Contents of Coffee Vending Machine for Black Only.

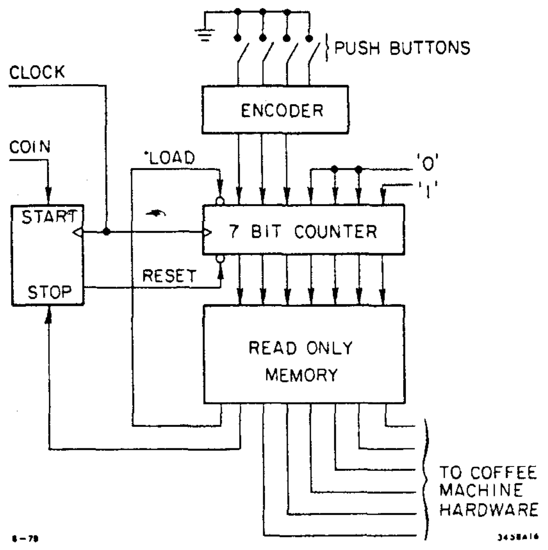


Figure 14: Microprogrammed Coffee Vending Machine with Multiple Sequences.

2.2 GENERAL MICROSEQUENCERS

The coffee machine sequence controller is an example of a microprogrammed processor. The processor's memory contains two fields; the load control bit and the other bits to control the hardware signals. A generalized version of this processor is shown in figure 15, where the encoder has been replaced by an instruction register. The OP-CODE field of the instruction register contains the high order bits of the starting address of a sequence. The push buttons of the coffee machine have been replaced by the machine instruction.

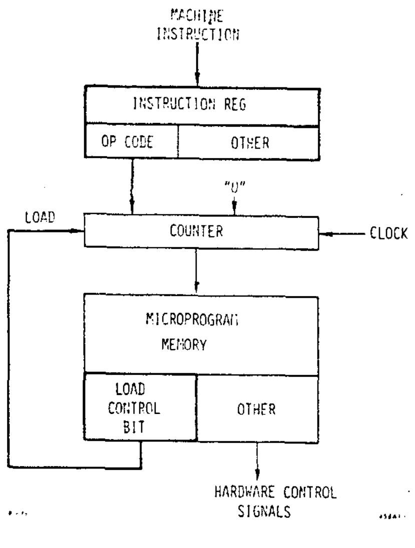


Figure 15: Basic Microsequencer.

In a more general processor, one may have sequences of widely different length and the circuit shown in figure 15 will lead to large areas of unused memory. The introduction of another memory, the MAPPING ROM, between the OP-CODE and the program counter will allow the flexibility of starting a sequence at any arbitrary address. The OP-CODE is used as the address of the MAPPING ROM and the output of the MAPPING ROM becomes the starting address for the program counter. This MAPPING ROM is shown in figure 16 and it is another example of using memory as a programmable decoder.

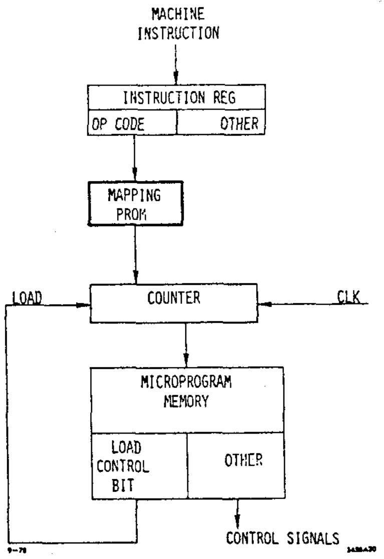


Figure 16: Microsequencer with MAPPING ROM.

With the microsequencer shown in figures 15 or 16, the flow of the program can only be the next sequential address until another sequence is started when the LOAD bit is present. This sort of flow is show schematically in figure 17 . At instruction 50 of the figure, for example, the next instruction can only be instruction 51. In this sequential flow the processor is said to execute the CONTINUE (CONT) instruction.

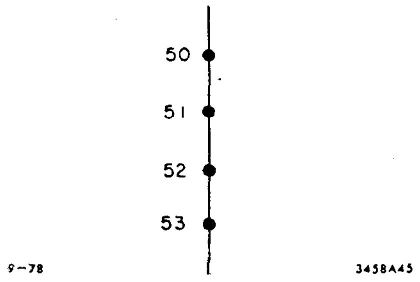


Figure 17: Continue Instruction.

One useful way to add flexibility to the microprocessor would be to allow the program the jump to an address which is contained in the microprogram. A method of doing this is shown in figure 18 . A multiplexer has been added between the output of the

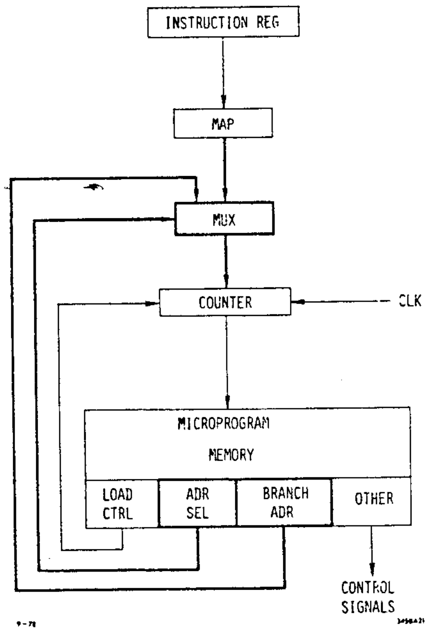


Figure 18: Microsequencer with JUMP logic.

MAPPING ROM and the input of the counter. One input of the multiplexer is the MAPPING ROM while the other comes from a part of the output of the microprogram memory. The latter is called the BRANCH ADDRESS field of the microprogram memory. One additional bit from the microprogram memory is routed to the SELECT input of the multiplexer so that when the bit is in one level the output of the Mapping ROM is routed to the input of the counter and when the bit is in the other level the BRANCH ADDRESS field of the micro-program memory is routed to the input of the counter. This bit is called the ADDRESS SELECT (ADR-SEL) field of the microprogram memory. The flow of the micro-program can now be altered as shown in figure 19 . After execution of instruction 53, the next instruction is 90. The processor is said to execute a JUMP (JMP) instruction at location 53.

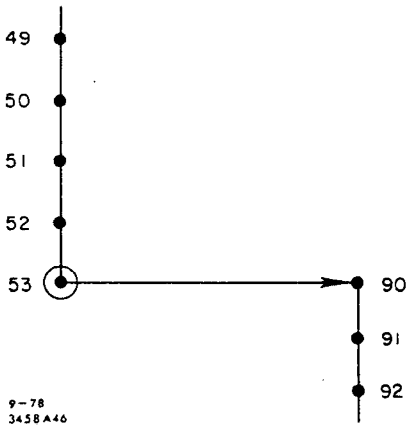


Figure 19: JUMP instruction.

A very important feature to add to this basic processor would be the ability to alter the flow of the program depending on the results of a previous operation. This is called CONDITIONAL BRANCHING and it can be implemented as shown in figure 20 . The LOAD input to the program counter is now taken from the output of multiplexer which is called the CONDITION CODE MULTIPLEXER. One of its inputs is selected by part of the output of the microprogram called the CONDITION CODE field. Note that one of the inputs to the multiplexer is a logic 'Ø'. When this input is selected, the LOAD input to the counter is always 'Ø' so that the counter goes to the next sequential address. Another input to the multiplexer is a logic '1'. When this input is selected, the counter will always be loaded. These two inputs are necessary in order that this processor can execute the CONTINUE and JUMP instructions, respectively. When the third input to the CONDITION CODE MULTI-PLEXER is selected, the counter will either go to the next sequential instruction if the conditional input is 'Ø' or be loaded if the conditional input is '1'. Thus we have added the CONDITIONAL BRANCH instruction to the processor. An example of this flow is shown in figure 21 at instruction 53.

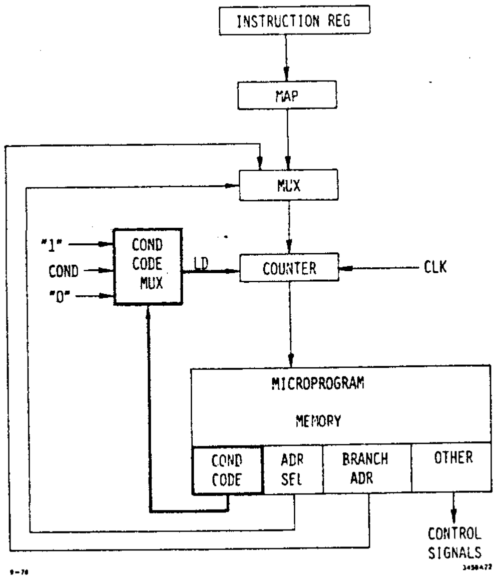


Figure 2Ø: Microsequencer with Conditional Branching.

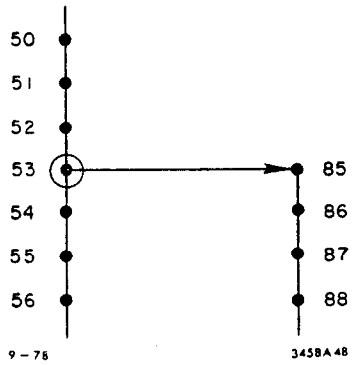


Figure 21: Conditional Branch Instruction.

At this point it is appropriate to take a look at the timing of the processor. Figure 22 shows the time sequence of the signals within the processor. Each microinstruction starts with the leading edge (i.e. the '0' to '1' transition) of the clock signal. When this signal is received by the microprogram counter, it increments its contents by one. The change of its output does not occur instantaneously, however. Each logic gate within the counter circuit has a response time called its "propagation delay". Thus it is only some time after the counter receives the clock signal that its output switches to the next address. For example, with a standard Schottky TTL counter the delay from clock to output is 13 nsec.

The microprogram memory also has a delay between the time an address is presented to its input and valid data is available at its output. This delay is called the "access time" of the memory and for Schottky TTL memories it is on the order of 50 nsec. The period of time from the generation of a new address until the output of a memory is steady is called the "fetch" time. Note that for the microprocessor we are studying, the total fetch time is equal to the sum of the propagation time of the counter and the access time of the microprogram memory.

After the fetch time, the process under control of the microprogram memory starts its execution. Again this process is only finished after a delay called the "execute time" which may be on the order of 100 nsec depending on what is being done. At the end of this period we have the results which may now be saved at the leading edge of the next clock signal in say an accumulator. Thus the minimum cycle time of the microprocessor is determined by the sum of the fetch and execute times. With the next edge of the clock signal the processor starts the next instruction.

Let us consider the microprocessor timing when a conditional branch instruction is executed. In the timing shown in figure 23 microinstruction i generates a result upon which we wish to conditionally branch. The result of this instruction is available during execution of microinstruction i+1, thus we should make microinstruction i+1 the conditional branch instruction. At the time of the third microcycle, we can start microinstruction i+2 or the instruction of the branch address depending on which path the result has taken us.

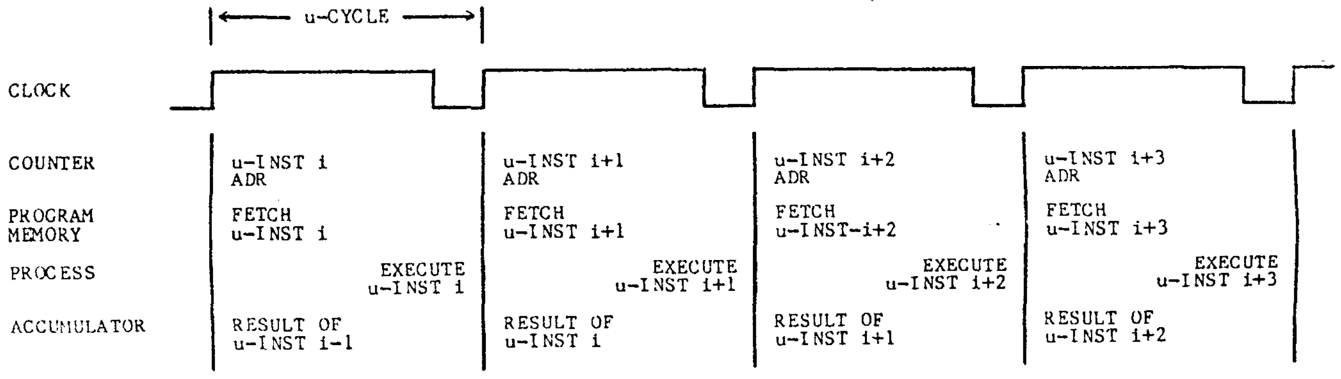


Figure 22: Sequential Timing with Program Counter.

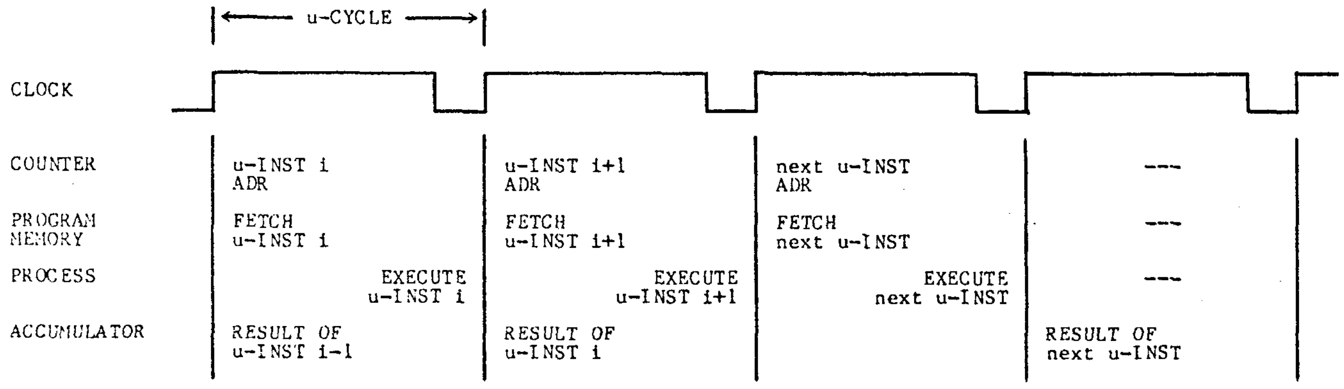


Figure 23: Conditional Branch with Program Counter.

One of the frequent requirements of logic systems in High Energy Physics is speed. With speed in mind, one could ask why do we use up a whole microcycle to do a branch instruction ? So let us consider for a moment how the timing would change if we attempted to do the conditional branch in the same microcycle as the execute. The condition upon which we want to branch would not be ready until the end of the execute time. It would be at that time that the condition would begin to propagate through the CONDITION CODE MULTIPLEXER and be presented to the LOAD input of the counter. Before the clock signal can be asserted at the counter, we must wait a period of time called the "set-up" time so that the counter can do the LOAD or COUNT function correctly. Using standard Schottky TTL circuits, the sum of the multiplexer propagation delay and the counter set-up time would add another 35 nsec to the minimum micro-cycle time. If the microcycle time is constant, then this additional time would be added to all micro-instructions whether they contained a branch or not. Thus depending on the number of branches in a program and the execute time, program execution time may be faster with the separate branch and execute instructions.

A much more important improvement can be made in program execution speed by using the technique of "pipelining". Note that in figure 22 that during the FETCH time the process under control is effectively idle since it is waiting for the output of the microprogram memory to become steady. Also during the execute time, the microprogram memory is effectively idle since it is merely holding its output steady for the execution. By inserting a register at the output of the microprogram memory as shown in figure 24, one can overlap or "pipeline" the fetch and execute times. One can see how this works by looking at the timing in figure 25 . With the leading edge of the first clock signal, the microprogram counter advances to microinstruction i and after the FETCH time the output of the micro-program memory is presented to the input of the PIPELINE REGISTER. As with the counter, one must wait a set-up time before the clock can be asserted to the register. When the clock does arrive, the microprogram memory output is stored in the PIPELINE REGISTER and after its propagation delay, the micro-instruction i is presented at the output of the register so that the execution of that instruction can begin. With this same clock edge, the micro-program counter advances to microinstruction i+1 and the FETCH of this instruction begins. Thus the FETCH of one instruction is done simultaneously with the EXECUTE of the previous instruction. The minimum microcycle time is now determined by the longer of the FETCH or EXECUTE times rather then the sum of them.

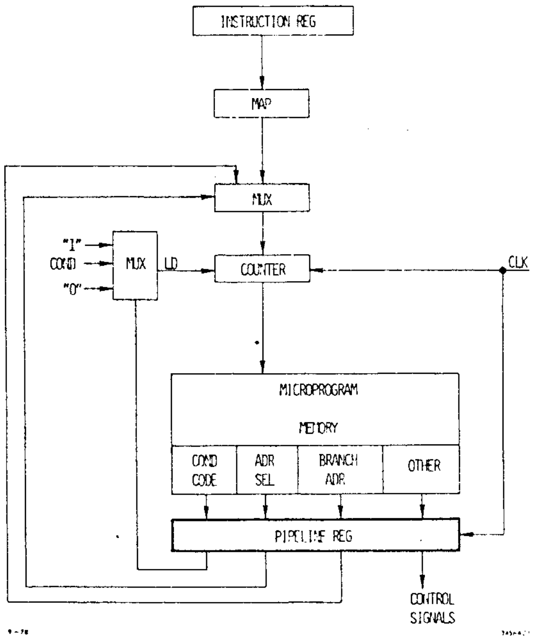
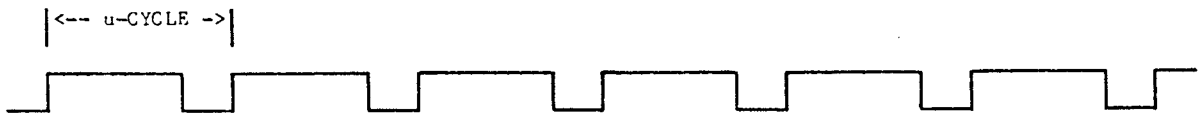


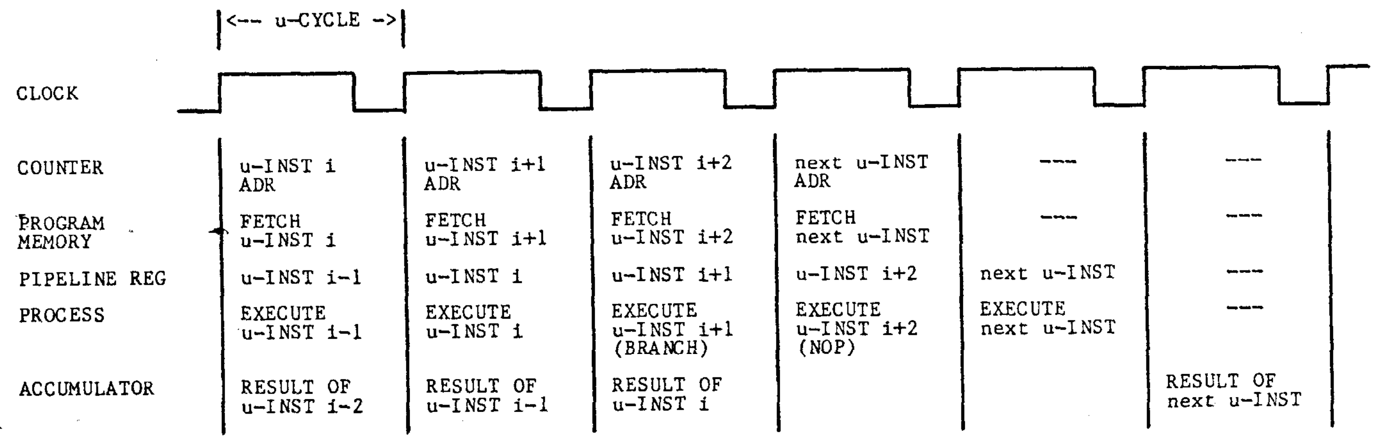
Figure 24: Microsequencer with Pipeline Register.



COUNTER	u-INST i ADR	u-INST i+1 ADR	u-INST i+2 ADR	u-INST i+3 ADR	u-INST i+4 ADR	u-INST i+5 ADR
PROGRAM	FETCH u-INST i	FETCH u-INST i+1	FETCH u-INST i+2	FETCH u-INST i+3	FETCH u-INST i+4	FETCH u-INST i+5
PIPELINE REG	u-INST i-1	u-INST i	u-INST i+1	u-INST i+2	u-INST i+3	u-INST i+4
PROCESS	EXECUTE u-INST i-1	EXECUTE u-INST i	EXECUTE u-INST i+1	EXECUTE u-INST i+2	EXECUTE u-INST i+3	EXECUTE u-INST i+4
ACCUMULATOR	RESULT OF u-INST i-2	RESULT OF u-INST i-1	RESULT OF u-INST i	RESULT OF u-INST i+1	RESULT OF u-INST i+2	RESULT OF u-INST i+3

Figure 25: Sequential Control with Program Counter and Pipeline Register.

The circuit shown in figure 24 leads to faster program execution. It has one difficulty with conditional branch instructions however. Consider a conditional branch on the results of instruction i as shown in the timing diagram in figure 26 . With the first microcycle we have the FETCH of micro-instruction i and with the second, we have the FETCH of microinstruction i+1 and the EXECUTE of micro-instruction i. The results of this instruction are ready to be tested in the third microcycle so clearly microinstruction i+1 should be the conditional branch instruction. At the end of the third cycle we begin the FETCH of the next microinstruction which is either microinstruction i+3 or the microinstruction located at the BRANCH ADDRESS. The problem is: what can the execution unit do during the fourth micro-cycle? The answer is that it can only do something which does not depend on which path the program has taken after the branch instruction. In most cases, nothing useful can be done by the execution unit during this cycle so that the microinstruction after the branch instruction (microinstruction i+2 in this case) becomes a NO OPERATION (NOP), which is a waste of execution time.



|<-- u-CYCLE ->|						
CLOCK						
COUNTER	u-INST i ADR	u-INST i+1 ADR	u-INST i+2 ADR	next u-INST ADR	---	---
PROGRAM MEMORY	FETCH u-INST i	FETCH u-INST i+1	FETCH u-INST i+2	FETCH next u-INST	---	---
PIPELINE REG	u-INST i-1	u-INST i	u-INST i+1	u-INST i+2	next u-INST	---
PROCESS	EXECUTE u-INST i-1	EXECUTE u-INST i	EXECUTE u-INST i+1 (BRANCH)	EXECUTE u-INST i+2 (NOP)	EXECUTE next u-INST	---
ACCUMULATOR	RESULT OF u-INST i-2	RESULT OF u-INST i-1	RESULT OF u-INST i			RESULT OF next u-INST

Figure 26: Condition Branch with Program Counter and Pipeline Register.

The circuit of the processor can be changed to fix this branching problem without slowing down the program execution as is shown in figure 27 . The microprogram counter has been replaced by an incrementer and a register, whose output is routed to an additional input to the address multiplexer, and the CONDITION CODE MULTIPLEXER has been replaced by some combinational logic. The address multiplexer is now routed directly to the address inputs of the microprogram memory. This multiplexer is called the NEXT ADDRESS MULTIPLEXER. An incrementer is a circuit whose output is equal to its input plus one. The new register is called the MICROPROGRAM COUNTER even though it is no longer a counter. In the circuit shown the current microprogram address plus one is stored into the MICROPROGRAM COUNTER with each clock edge. Thus when the MICROPROGRAM COUNTER is selected as the output of the NEXT ADDRESS MULTIPLEXER, one has the CONTINUE instruction in effectively the same way as when we forced a COUNT of the counter in figures 15, 16, 18, 20, and 24 . A jump instruction is accomplished by selecting the BRANCH ADDRESS as the output of the NEXT ADDRESS MULTIPLEXER and a conditional branch instruction is accomplished by selecting either the BRANCH ADDRESS or the MICROPROGRAM COUNTER depending on the state of the CONDITION input.

The timing of this circuit for non branching instructions appears to be the same as the previous one as is shown in figure 28 . The minimum microcycle time is still determined by the longer of the FETCH or EXECUTE times. But the conditional branch timing is much improved as is shown in figure 29 . With the first microcycle in figure 29 we have the FETCH of microinstruction i and with the second we have the FETCH of microinstruction i+1 and the EXECUTE of microinstruction i as before. With the third microcycle we begin to test the results of microinstruction i so clearly microinstruction i+1 is the branch instruction. The condition input selects the next address during the third microcycle so that the FETCH of the next microinstruction begins and ends during this cycle. Thus with the leading edge of the clock of the fourth microcycle, the next

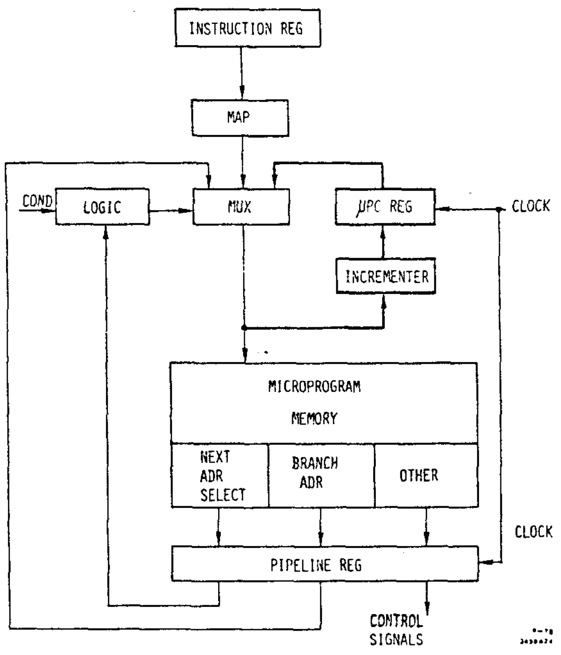
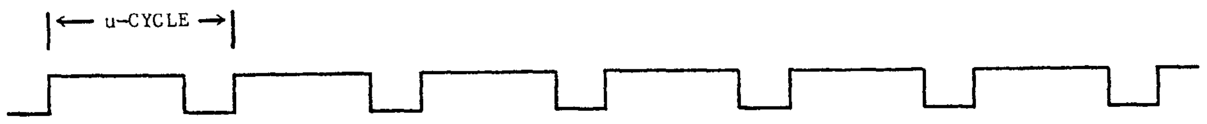


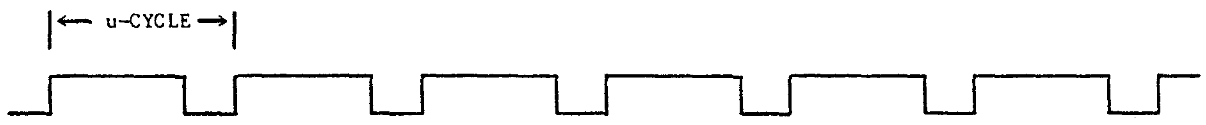
Figure 27: Microsequencer with Microprogram Counter and Incrementer.

microinstruction is stored into the PIPELINE REGISTER and the EXECUTE of this instruction begins. At the same time the next+1 instruction address is stored into the MICROPROGRAM COUNTER so that the FETCH of this instruction can begin. Thus even with a conditional branch instruction, this circuit for a microprocessor makes efficient use of microcycle time. It is this type of circuit that we will find to be commercially available as a LSI microcircuit when we study microsequencers.



PROGRAM MEMORY	FETCH u-INST i	FETCH u-INST i+1	FETCH u-INST i+2	FETCH u-INST i+3	FETCH u-INST i+4	FETCH u-INST i+5
PIPELINE REG	u-INST i-1	u-INST i	u-INST i+1	u-INST i+2	u-INST i+3	u-INST i+4
PROCESS	EXECUTE u-INST i-1	EXECUTE u-INST i	EXECUTE u-INST i+1	EXECUTE u-INST i+2	EXECUTE u-INST i+3	EXECUTE u-INST i+4
ACCUMULATOR	RESULT OF u-INST i-2	RESULT OF u-INST i-1	RESULT OF u-INST i	RESULT OF u-INST i+1	RESULT OF u-INST i+2	RESULT OF u-INST i+3

Figure 28: Sequential Control with Program Counter Register.



PROGRAM MEMORY	FETCH u-INST i	FETCH u-INST i+1	FETCH next u-INST	---	---	---
PIPELINE REG	u-INST i-1	u-INST i	u-INST i+1	next u-INST	---	---
PROCESS	EXECUTE u-INST i-1	EXECUTE u-INST i	EXECUTE u-INST i+1 (BRANCH)	EXECUTE next u-INST	---	---
ACCUMULATOR	RESULT OF u-INST i-2	RESULT OF u-INST i-1	RESULT OF u-INST i		RESULT OF next u-INST	---

Figure 29: Conditional Branching with Program Counter Register.

3. AN EXAMPLE: A SIMPLE SCANNER

With the next address instructions we have defined so far it is already possible to look at a practical example. Suppose one had a large set of devices which potentially contain a data point on a given event, but for a given event let us say that only a small fraction of the devices have any valid data. Let us design a controller which would scan the devices for data and store the data into a buffer memory along with the address of the devices with data. Figure 30 shows a block diagram of the proposed scanning set-up. For simplicity let each device be interrogated (or addressed) by a signal sent on a cable which we will call the NEXT signal. If the device had data it would send back the data on a bus along with a response signal which we will call DATA-VALID. If the device did not have data it would send a '0' on the DATA-VALID bus. The NEXT signal would be daisy chained from device to device so that after each device received one NEXT signal it would pass the next NEXT signal to the next device until the system was reset for another scan. The NEXT signal from the last device on the chain is routed back to the scanner where it is called the DONE signal so the scanner knows when to stop.

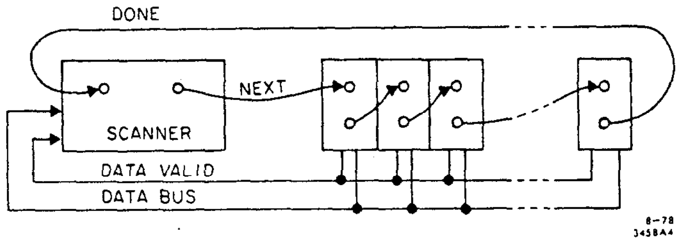


Figure 30: Block Diagram of Scanner Set-up.

For the moment we shall ignore all details on how the data is read into the devices, read out of the buffer memory, etc. The scanning processor should have two internal counters, a DEVICE COUNTER (D.C.) to keep track of which device has valid data, if any, and an ADDRESS COUNTER (A.C.) to point to the next buffer memory address to be filled. The sequence of events the scanning processor should follow is shown in the flow chart given in figure 31 . When the processor starts, it should first reset the DEVICE COUNTER, reset the ADDRESS COUNTER, and send the first NEXT signal as shown in step 0. In the next step, it can test the DATA VALID signal to see if any data was found in the first device. If it is false, it should then test the DONE signal (step 2). If this signal is also false, it can proceed to the next device by incrementing the DEVICE COUNTER and sending another NEXT signal. The processor continues by going back to step 1, in order to see if any data is found in that device. If data is found, it can write the data along with the contents of the DEVICE COUNTER into the buffer memory (step 4). Then it may increment the ADDRESS COUNTER (step 5) and try the next device by going to step 3. The sequence continues until the DONE signal is detected at step 2, in which case the process goes into a STOP state at step 6.

Figure 32 shows a possible implementation of a microprogrammed processor to perform this task. The microprogram memory contains five fields as shown in figure 33 . Let us examine each field in order to understand how the processor works. The first two fields control the next address of the processor. The first field is the BRANCH ADDRESS field (bits 0-3), which is four bits wide. These four bits are routed from the output of the PIPELINE REGISTER to the '1' input of the NEXT ADDRESS MULTIPLEXER as shown in figure 32 . Whenever the SELECT input of this multiplexer is a logic '1', the BRANCH ADDRESS becomes the address input to the microprogram memory. The next field is the ADDRESS SELECT field which is two bits wide and it controls the CONDITION CODE MULTIPLEXER. The conditions '0' and '1' are inputs to this multiplexer in order that CONTINUE and JUMP next address instructions may be executed. That is, when the ADDRESS SELECT field is '00', the '0' input is selected and the next address will always come from the MICROPROGRAM COUNTER. When the ADDRESS SELECT field is '11', the '1' input is selected and the next address will always come from the BRANCH ADDRESS field of the PIPELINE REGISTER. The two conditions which must be tested to control the program flow, DATA VALID and DONE, are also inputs to this multiplexer. When the ADDRESS SELECT field is '01', the DATA VALID signal is selected and the state of this signal will determine whether the

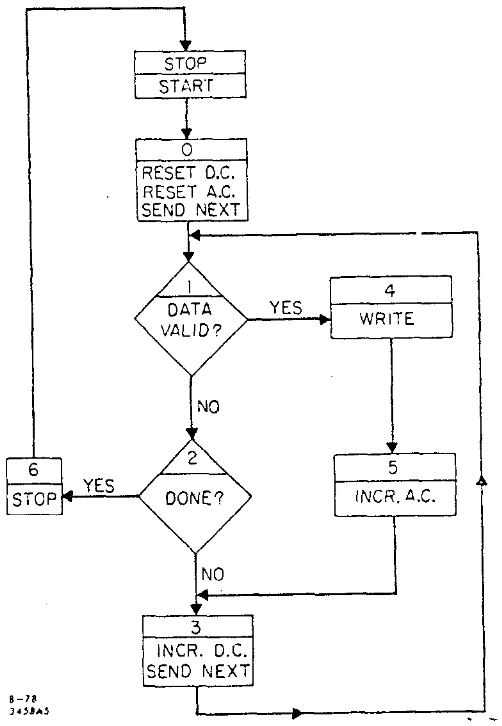


Figure 31: Flow Chart of Scanning Processor.

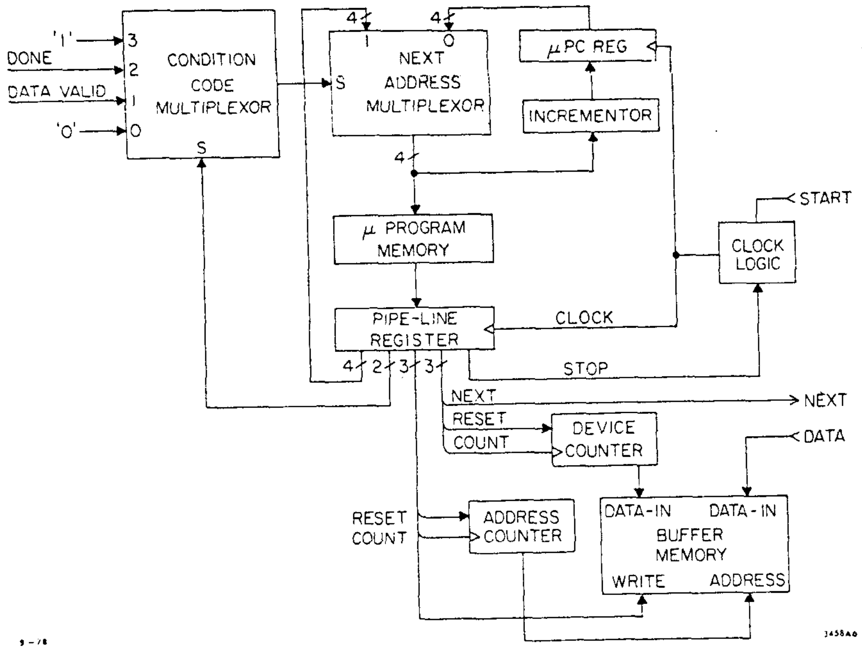


Figure 32: Block Diagram of Simple Scanner

Bits				
0-3	4-5	6-8	9-11	12
Branch Address	Condition Code Multiplexer	Buffer Memory Control	Device Counter Control	End

The fields have the following meaning

Bits 0-3: BRANCH ADDRESS for Branch or Jump instruction.

Bits 4-5: CONDITION CODE MULTIPLEXER Control

00	CONTinue	next address is PROGRAM COUNTER REGISTER.
01	BRDV	If DATA-VALID is '1', next address is from BRANCH ADDRESS.
00	BRDONE	If DONE is '1', next address is from BRANCH ADDRESS.
11	JUMP	next address is from BRANCH ADDRESS.

Bit 6: Resets buffer memory ADDRESS COUNTER.

Bit 7: Increments buffer memory ADDRESS COUNTER.

Bit 8: Sends WRITE to buffer memory.

Bit 9: Generates NEXT signal.

Bit 10: Increments DEVICE COUNTER.

Bit 11: Resets DEVICE COUNTER.

Bit 12: Stops Scanner.

Figure 33: Definition of Simple Scanner Microinstructions.

MICROPROGRAM COUNTER or the BRANCH ADDRESS field is used as the next microprogram address. The same holds true when the DONE signal is selected when the ADDRESS SELECT field is '1Ø'.

The next 3 bits (bits 6-8) is the BUFFER MEMORY CONTROL field. Each bit of the field is wired to a control point of the buffer memory: if bit 6 is '1' the ADDRESS COUNTER is reset to zero; if bit 7 is '1' the ADDRESS COUNTER is incremented; and if bit 8 is '1' the contents of the DEVICE COUNTER and the DATA lines are written into the buffer memory.

The next 3 bits (bits 9-11) is the DEVICE CONTROL field. Again each bit of this field is wired to a control point dealing with the devices: if bit 9 is '1' a NEXT signal is sent out; if bit 1Ø is '1' the DEVICE COUNTER is incremented; and if bit 11 is '1' the DEVICE COUNTER is reset to zero. The last field contains only one bit (bit 12), when it is '1' the processor stops.

The program can now be written to perform the scan operation. Let us go through the flow chart shown in figure 31 again. At step Ø, we want to reset the ADDRESS COUNTER so bit 6 should be '1'. We don't want to increment this counter or write to the memory so bits 7 and 8 should be 'Ø'. Thus the BUFFER MEMORY CONTROL field should be set to '1ØØ'. The DEVICE COUNTER should be reset and a NEXT signal sent out, so the DEVICE CONTROL field should be set to '1Ø1'; i.e. bits 9 and 11 should be set to '1' and bit 1Ø to 'Ø'. The next microprogram address can be the next sequential instruction, so the ADDRESS SELECT field should be set to 'ØØ' in order to execute the CONTINUE next address instruction. It doesn't matter what the BRANCH ADDRESS field is set to since they are not used for the CONTINUE instruction, so we set it to 'ØØØØ'. Thus the contents of the first microprogram location (address ØØØØ) should be as shown in figure 34 .

	0-3	4-5	6-8	9-11	12
micro prog. addr.	Branch Address	Cond. Code Mult.	Buffer Memory control	Device counter control	End
0000	0000	00	1 0 0	1 0 1	0
0001	0100	01	0 0 0	0 0 0	0
0010	0110	10	0 0 0	0 0 0	0
0011	0001	11	0 0 0	1 1 0	0
0100	0000	00	0 0 1	0 0 0	0
0101	0011	11	0 1 0	0 0 0	0
0110	0000	11	0 0 0	0 0 0	1
0111			unused		
.					
.					
1111					

Figure 34: Program of Simple Scanner.

The next step tests the DATA VALID signal and leaves the buffer memory and device control alone. Thus both the BUFFER MEMORY CONTROL and DEVICE CONTROL fields both contain 'ØØØ'. The ADDRESS SELECT field is set to 'Ø1' to route the state of the DATA VALID signal to the NEXT ADDRESS MULTIPLEXER. The BRANCH ADDRESS field is set to the next micro-program address if DATA VALID signal is present which we will set to 'Ø1ØØ', i.e. step 4. Thus the contents of microprogram memory at location ØØØ1 should be as show in figure 34 .

At microprogram location ØØ1Ø, which will be the instruction if the DATA VALID tests fails, we should test the state of the DONE signal. The only changes from the previous instruction is that the ADDRESS SELECT field needs to be set to '1Ø' and another BRANCH ADDRESS needs to be chosen. Thus, if location Ø11Ø is to correspond to step 6 on the flow chart (figure 31), the contents of the microprogram memory at location ØØ1Ø should be as shown in figure 34 .

At microprogram location ØØ11, which will be the next instruction if the DONE test fails, we should increment the DEVICE COUNTER, send a NEXT signal, and go back to the DATA VALID test at location 1. Thus we set bits 9 and 1Ø to '1', the ADDRESS SELECT field to '11' and the BRANCH ADDRESS field to 'ØØØ1' as shown in figure 34 .

The remaining steps are programmed in a similar fashion. The whole program is summarized in figure 34 . Only six microprogram instructions were needed to program this processor to control the simple scanning operation. Locations 7 through 15 of the microprogram memory are unused.

4. WHY MICROPROGRAM

Up to this point, we have studied the basics of microprogramming and a simple example. We are now ready to evaluate whether we should or should not build our logic subsystems using the microprogramming technique. Keep in mind that most logic subsystems have two parts: one is the part under control and the other is the part which generates the timing sequences. One can identify these parts in logic systems as small as the simple scanner or as large the central processing unit of a major computer system.

An advantage in a microprogrammed system is its very clean and orderly structure. The simple scanner described in the previous section is an example. All control points within the processor are controlled by the microprogram memory. All registers, counters etc., can be synchronized by the same clock which makes it much easier to find faults in the circuit. The system is very easy to describe and document.

There are still other advantages, especially for larger, more complex logic systems. One advantage is the ease in which changes can be made. Suppose, for example, that one wanted to change the simple scanner so that it would stop scanning if the buffer memory became full. From the ADDRESS COUNTER one could obtain a signal indicating that the maximum buffer memory address had been attained. This signal could be routed to an additional input to the CONDITION CODE MULTIPLEXER and the program could be changed so that the buffer full condition would be tested after the ADDRESS COUNTER was incremented. Although this change would require changing some circuits such as the CONDITION CODE MULTIPLEXER, the change is quite simple to understand and implement compared to changing a hard wired or random logic design for the simple scanner. The more complex a logic subsystem, the easier it is to change a microprogrammed circuit compared to its random logic counterpart.

It is also much easier to add special features to a microprogrammed logic system. One desirable feature, for example, would be self diagnostic programs in the microprogram memory. These features probably require only additional memory space and a few extra circuits. But for a random logic design, the additional logic required to perform diagnostics may be as complex or difficult as the original logic itself.

There will be cases, however, where the random logic design is more suitable for a logic subsystem. The microprogrammed system may be slower, for example. The random logic design avoids the difficulty of programming the ROM. But if one is to use the LSI microcircuits, which we will be studying in the following sections, one must use the microprogramming technique.

5. LSI MICROCIRCUITS

There is an increasing number of LSI microcircuits becoming available. They all fit very well into a microprogrammed architecture. This is clearly becoming the "nouveaux vague" in bipolar technology. Most of these components are intended to be used in building computers and computer peripherals where the speed of bipolar circuits is necessary. In our field of High Energy Physics, we are not in the business of building computers but these microcircuits can nevertheless simplify how we build many of our logic systems.

The circuits can be classified by what part of a computer they are intended to be used. The following list shows some of the computer parts that currently available as LSI microcircuits:

Arithmetic/Logical Elements
Microprogram Control
Bipolar Memory
Interrupt Control
Input/Output Elements
Direct Memory Access Control
Timing Control
Shifting Elements
Status Storage and Multiplexing
Memory Address Control
Program Logic Arrays

For the logic designer in High Energy Physics, one would layout a sketch of the control task he wants to build. Then scan through the semiconductor catalogs to find which circuits have the capabilities of parts of his logic diagram. Frequently, one would find that only a fraction of an LSI circuit is needed to implement his circuit and it would seem wasteful to use it. On the other hand the cost of implementing the logic in traditional SSI and MSI may exceed the cost of this LSI circuit and the amount of space required on the circuit board may be reduced by using the LSI microcircuit.

There is clearly not enough time in these lectures to cover all of the above kinds of LSI microcircuits. We will therefore study only two of them, the microprocessor slice and the microsequencer. These two in some ways are the most interesting and the most different from the older SSI and MSI circuits.

5.1 THE BIPOLAR MICROPROCESSOR SLICE

The microprocessor slice is designed to be the principal arithmetic/logic element within the central processing unit of a computer or peripheral controller. The most widely used microcircuit of this type is the 2901A. It was introduced by Advanced Micro Devices in the summer of 1975 and has since been manufactured by most of the bipolar semiconductor manufacturers. Figure 35 shows a block diagram of this circuit. All the data paths shown are four bits wide. To form a processing unit with a larger number of data path bits, the circuit has appropriate inputs and outputs to allowed it to be cascaded with other slices. Thus four such slices can form a unit with 16 bits of data path and eight slices forms a unit with 32 bits of data path.

Let us start studying this microcircuit with its arithmetic logic unit (ALU). An ALU performs arithmetic and logical operations on two inputs, called R and S in figure 35, and produces an output which is called F. The ALU of the 2901 can perform one of eight functions on the operands R and S. The function is selected by three input signals which form a three bit function code which is part of the microinstruction of the circuit. That is, the user provides a three bit binary number, from 0 through 7, on three input pins of the circuit to control which function is to be performed. There are three arithmetic functions:

R PLUS S,
S MINUS R, and
R MINUS S;

and five logical functions:

R OR S,
R AND S,
NOT R AND S,
R EXCLUSIVE-OR S, and
R EXCLUSIVE-NOR S.

The R and S operands are each outputs of separate multiplexers. The R MULTIPLEXER has as inputs the DIRECT DATA inputs (D), the output of the A-LATCH, or

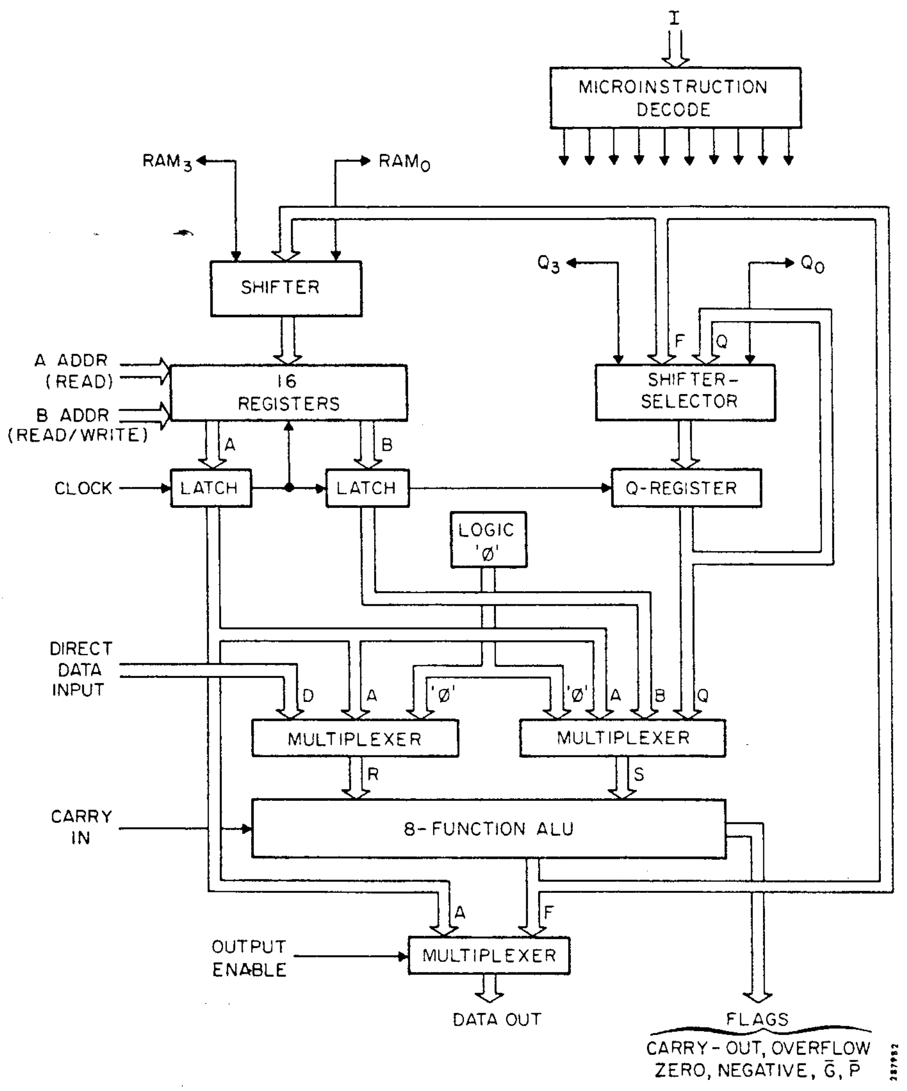


Figure 35: Block Diagram of 29Ø1A Microprocessor Slice.

a binary 'Ø'. The D input allows the user to bring in data from outside the circuit, say from memory, so four pins of the circuit is used for this purpose. The significance of the A-LATCH and the usefulness of providing a 'Ø' as one of the inputs will be seen in the following paragraphs. The S-MULTIPLEXER has for its input also a binary 'Ø' and the A-LATCH as well as the output of a B-LATCH and the contents of a Q-REGISTER. With these inputs there are twelve combinations possible for the R and S operands. Twelve is not a nice number, so the manufacturer has chosen eight of these twelve that he feels are the most useful. That is, three bits of the microinstruction are used to select which combination of the R and S inputs are used as operands to the ALU. These three bits is called the Source Code of the microinstruction.

The Source Codes can be best seen in a matrix of the Source Code versus the Function Code as is shown in figure 36 . Note that the entries of this matrix, look very much like computer operations, i.e. A PLUS B, A OR B, D MINUS A, B PLUS 1, etc. It is essentially a sufficient set to allow the user to do any operation on two binary or logical quantities. Note also that some of the elements of this matrix have two entries. The difference between these entries is whether the CARRY-IN to the least significant bit of the ALU is '1' or 'Ø'. For the addition of two numbers the CARRY-IN should be 'Ø', but for subtraction in 2's complement form, the CARRY-IN to the least significant bit should be '1'. Thus we realize that in order to control this microcircuit we must provide every small detail to get what we want. We are truly programming at the microinstruction level.

Let us now return to the A- and B-LATCHes. Within the 29Ø1A there are 16 registers which are organized as a "dual port memory". That is, two addresses can be read from the memory simultaneously. The 16 word memory is called a register file and the user provides a four bit address to select one of 16 words in the register file for each port of the memory. These are called the A and B addresses. The clock signal input to the circuit is used to hold the outputs from these two ports in a special kind of register called a latch. Thus the output of the A- and B-LATCHes are the contents of the register at the A and B addresses when the clock signal makes the transition from low to high.

The are two sets of results from the ALU. The first is called F in the figure 35 and it is the result of the function on the two operands. The other is a set of status conditions indicating if the operation resulted in a carry out of the most significant bit, a 2's complement arithmetic overflow, a zero result, or a negative result. In addition to these status signals the ALU incorporates the standard Carry-Look-Ahead techniques to speed up operations over many slices which requires the generation of the Carry Generate (G) and Carry Propagate (P) signals.

	$I_{2,1,0}$ Octal	0	1	2	3	4	5	6	7
Octal $I_{5,4,3}$	ALU Source / ALU Function	A, Q	A, B	O, Q	O, B	O, A	D, A	D, Q	D, O
0	C_n = L R Plus S	A + Q	A + B	Q	B	A	D + A	D + Q	D
	C_n = H	Q + 1	A + B + 1	Q + 1	B + 1	A + 1	D + A + 1	D + Q + 1	D + 1
1	C_n = L S Minus R	Q - A - 1	B - A - 1	Q - 1	B - 1	A - 1	A - D - 1	Q - D - 1	- D - 1
	C_n = H	Q - A	B - A	Q	B	A	A - D	Q - D	- D
2	C_n = L R Minus S	A - Q - 1	A - B - 1	- Q - 1	- B - 1	- A - 1	D - A - 1	D - Q - 1	D - 1
	C_n = H	A - Q	A - B	- Q	- B	- A	D - A	D - Q	D
3	R OR S	$A \vee Q$	$A \vee B$	Q	B	A	$D \vee A$	$D \vee Q$	D
4	R AND S	$A \wedge Q$	$A \wedge B$	0	0	0	$D \wedge A$	$D \wedge Q$	0
5	$\overline{R}$ AND S	$\overline{A} \wedge Q$	$\overline{A} \wedge B$	Q	B	A	$\overline{D} \wedge A$	$\overline{D} \wedge Q$	0
6	R EX-OR S	$A \forall Q$	$A \forall B$	Q	B	A	$D \forall A$	$D \forall Q$	D
7	R EX-NOR S	$\overline{A \forall Q}$	$\overline{A \forall B}$	$\overline{Q}$	$\overline{B}$	$\overline{A}$	$\overline{D \forall A}$	$\overline{D \forall Q}$	$\overline{D}$

+ = PLUS; $-$ = MINUS; $\vee$ = OR; $\wedge$ = AND; $\forall$ = EX-OR.

8 - 78
3458A3

Figure 36: Source Operand and ALU Function Matrix.

MICRO CODE				RAM FUNCTION		Q-REG. FUNCTION		Y OUTPUT	RAM SHIFTER		Q SHIFTER	
I_8	I_7	I_6	Octal Code	Shift	Load	Shift	Load		RAM_0	RAM_3	Q_0	Q_3
L	L	L	0	X	NONE	NONE	F → Q	F	X	X	X	X
L	L	H	1	X	NONE	X	NONE	F	X	X	X	X
L	H	L	2	NONE	F → B	X	NONE	A	X	X	X	X
L	H	H	3	NONE	F → B	X	NONE	F	X	X	X	X
H	L	L	4	DOWN	F/2 → B	DOWN	Q/2 → Q	F	F_0	IN_3	Q_0	IN_3
H	L	H	5	DOWN	F/2 → B	X	NONE	F	F_0	IN_3	Q_0	X
H	H	L	6	UP	2F → B	UP	2Q → Q	F	IN_0	F_3	IN_0	Q_3
H	H	H	7	UP	2F → B	X	NONE	F	IN_0	F_3	X	Q_3

9 - 78
3458A42

Figure 37: 2901A Destination Codes.

Generally, one would like to do something with the results of the ALU operation. The data paths within the 2901 provide us with many possibilities. First, the results may be written back into the register file. At this point we start running out of pins available on the circuit package, so the manufacturer has made a compromise in that when results are written to the register file they are written into the register of the B address. But before writing into the B address the 2901A has the capability of shifting the results to the right or to the left. It can also write the results to the Q-REGISTER. We can also output the results on four pins of the package or we can use these output pins to output the contents of the A-LATCH. Of all the possibilities the manufacturer has again used 3 pins on the package to allow us to select one of eight possibilities. These 3 bits are called the Destination Code of the microinstruction and these codes are shown in figure 37 . Note that one of the Destination Codes doesn't write the results anywhere (Code 1). It is a No-Operation code (NOP) and is useful when we want to do a COMPARE operation without destroying the contents of any register. Note also that there are Destination Codes where the Q-REGISTER is shifted while shifting the results written into the register file. The purpose of this code is to allow the user to program the circuit to do multiplication, division, and double length shifts and rotates.

In summary, a microprocessor slice can form the core of a central processor unit within a computer. It is controlled by providing it with a microinstruction which it uses internally. In the case of the 2901A, the microinstruction must be at least 18 bits in length: 3 bits for the Source Code, 3 bits for the Function Code, 3 bits for the Destination Code, 1 bit for the CARRY-IN to the least significant bit and four bits each for the A and B addresses. Although the 2901A is the most widely used microprocessor slice today, there are others on the market which may be more suitable in certain applications. Table 1 lists all the microprocessor slices currently available with a few comments on their individual features. For more details, the reader is referred to the specification data sheets from the various manufacturers, or to some of the numerous articles in some of the journals[1,2].

5.2 <u>AN EXAMPLE WITH THE 2901A</u>

An example of the use of the 2901A is a microprocessor called the 168/E which was developed by my colleagues and me at S.L.A.C. Figure 38 is a block diagram of the processor. As with other designs using it, the 2901 forms the core of the data processing and there are circuits around it to form a complete microprocessor. For the 168/E, the choice of the circuits around the 2901 was based on what would be easy to program. For this purpose some assembly code written for the IBM 370 computer was used as a model for the kind of operations that would

TABLE 1

List of Microprocessor Slices Currently Available.

Part no.	Originated by	Second Source	(bits)	Tech.	Comments
2901A	Advanced Micro Devices	Fairchild, Monolithic Memories, National, Raytheon, Motorola, and Signetics.	4	S-TTL	most popular slice, dual-port architecture
2903	Advanced Micro Devices.	National.	4	S-TTL	Improved version on 2901A, expandable register file, built-in multiply, divide, etc.
6701	Monolithic Memories.	ITT	4	S-TTL	similar to 2901A
3002	Intel	Signetics	2	S-TTL	accumulator orientated
9405	Fairchild	Signetics	4	S-TTL C-MOS	smaller package
74S481	Texas Inst.	none	4	S-TTL	similar to 2903, but has no register file
10800	Motorola	Fairchild	4	ECL	has no register file
SBP0400	Texas Inst.	none	4	IIL	very slow

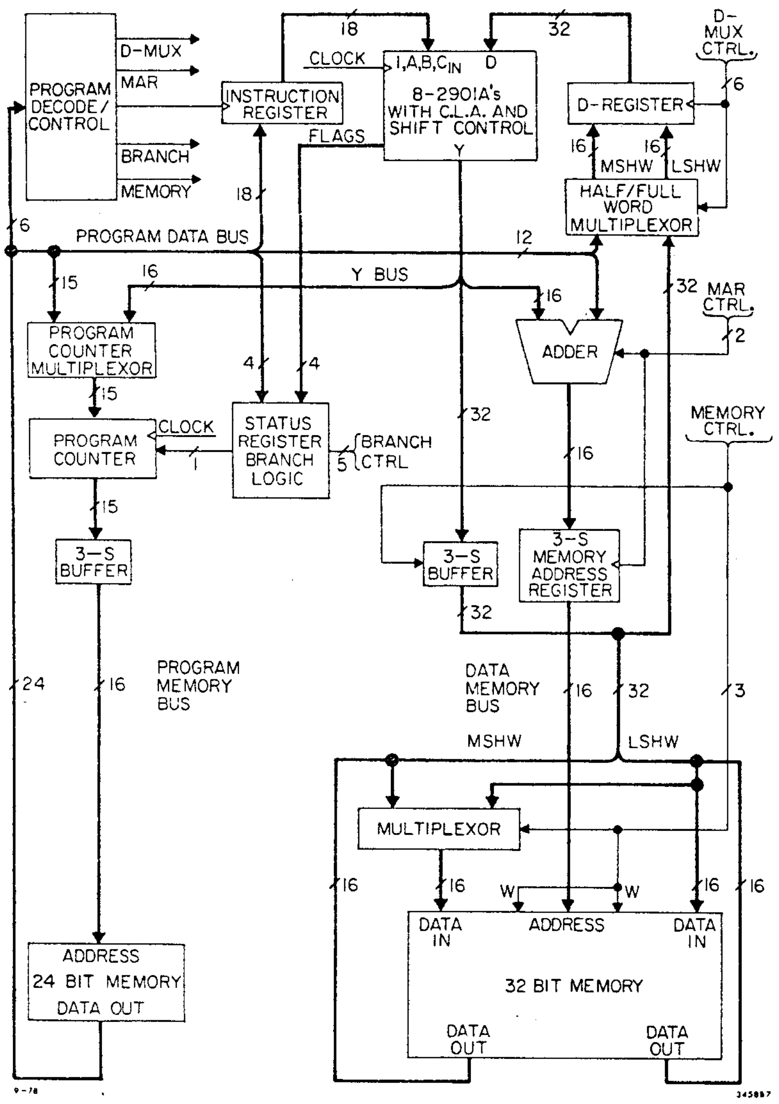


Figure 38:

Block Diagram of 168/E Microprocessor.

be necessary to have a useful processor. An example of a typical DO LOOP found in many of the programs we wished the microprocessor to be able to handle is shown below.

```
LOOP     C     Ø,ED(9,1Ø)
         BL    GOTE
         SR    9,1
         BNM   LOOP
```

Let us consider how this piece of code is executed on the 168/E in order to understand the functions of the circuits around the 29Ø1A microprocessor slices. The first IBM instruction is a memory to register comparison. The memory address is calculated as a sum of the contents of registers 9 and 1Ø plus the contents of a 12 bit displacement field (ED) which is part of the instruction. The IBM 37Ø has 16 registers and so does the 29Ø1A. Thus a one to one identification of the registers in the 37Ø with those of the 168/E was made. We have already studied the 29Ø1A well enough to see how one can add the contents of two registers together and output the sum. This would be the first step the 168/E should do in order to follow the example of the IBM 37Ø program. That is, a 168/E microinstruction with the 29Ø1A Source Code 1, Function Code Ø, and Destination Code 1 as can be seen from figures 36 and 37 .

The next step would be to add to this sum the 12 bits of the displacement field. The 168/E performs this operation in a separate microinstruction with an additional Adder circuit. In this microinstruction, 12 bits of the microprogram memory are routed to one input to the adder. The other half of the adder is connected to the output of the 29Ø1A. The sum of this addition is strobed into a MEMORY ADDRESS REGISTER whose output is connected to the address inputs of the data memory. The access time of the data memory is fast enough for the data memory output to be strobed into the D-REGISTER at the end of the microcycle.

The last step of the COMPARE instruction is to make the comparison between memory, which has now been strobed into the D-REGISTER of the 168/E, and the contents of register Ø. This step is performed by another microinstruction in which the instruction for the 29Ø1A uses its D input as one of the ALU source operands, i.e. Source Code 5, Function Code 1, and Destination Code 1.

The next IBM 37Ø instruction is a conditional branch in which the next instruction is to be taken from an address labeled 'GOTE', if the result of the comparison was negative. Thus to the 168/E microinstructions we have already defined, we must add conditional branch microinstructions. The 168/E uses a binary counter to control the next microinstruction instruction address. As in figure 2Ø, the flow of the program execution can be altered by asserting a signal on the LOAD pin of the counter circuit. To control the conditions with which one wants to branch, the IBM 37Ø is again used as a model. The conditional branch instruction of the IBM 37Ø has a 4 bit field called the mask which specifies what conditions are required to force a branch. The conditions after arithmetic operations are a zero result, a negative result, a positive result, and a arithmetic overflow result. The arithmetic instructions sets one of these conditions to be true after the operation. If the condition which was set matches with a set bit in the mask of the conditional branch instruction, then the branch is taken, otherwise the next sequential instruction is executed.

The status outputs of the 29Ø1A do not exactly correspond to those of the IBM 37Ø but with a few logic circuits one can produce identical codes. Thus the 168/E conditional branch microinstruction has been set up with the same 4 bit mask as the IBM 37Ø. If there is a match between the 4 bit mask and the modified 29Ø1A status bits then the program counter is put into the LOAD state. Fifteen bits from the microprogram memory contain the branch address which is routed to the parallel load inputs of the program counter.

The rest of the instructions of the DO LOOP can be emulated by the 168/E with the microinstructions already described. The structure of the 168/E is typical of structures in which the 29Ø1A forms the core of data processing. We can identify the parts which are under control and the part which provides the timing sequences. The result of the structure chosen among the many that were possible is that it easy to translate the IBM 37Ø instructions into the microinstructions of the 168/E. In fact if one now looks at the list of the primary IBM 36Ø/37Ø instructions, as shown in table 2, one sees that an impressive number of the instructions can be exactly emulated by the 168/E. These instructions turn out to be about the same subset of the instructions that the IBM FORTRAN H complier generates when dealing with 2 or 4 byte integer or 4 byte logical variables. The Floating Point instructions are executed in a separate processing unit not show in figure 38 . Thus the 168/E microprocessor can be programmed in FORTRAN by using the IBM FORTRAN compiler to generate machine instructions, then using a program which runs on a IBM computer to translate these machine instructions into the microinstructions of the 168/E. This is possible because the 29Ø1A has the same number of registers as the IBM 36Ø/37Ø and it can perform all the integer arithmetic and logical operations of the IBM 36Ø/37Ø (in fact, it can do some operations that the IBM computer can not). Also, the circuitry around the 29Ø1A makes a processor with the same form of memory addressing and conditional branching. Thus either the 29Ø1A with some circuitry around it forms a very powerful microprocessor or the IBM 36Ø/37Ø is a very simple computer depending on your point of view.

We have already mentioned speed of execution as one of the frequent requirements for logic systems in High Energy Physics. So one can ask what is the speed of the 168/E? The 168/E was implemented with mostly Low Power Schottky circuits which have a typical gate propagation delay of 5 nsec, yet the speed of execution of a program is only between 1.3 and 1.8 times slower than the IBM 37Ø/168. Compared with typical minicomputers, the 168/E speed is about 3 to 1Ø times faster. And what about the cost? The main cost of the 168/E processor is the eight 29Ø1As which is about 15Ø US$. The other circuits, circuit board, sockets, and power supply add less than another 3ØØ US$. Thus the user of these LSI microcircuits can build for himself very powerful and fast processors with standard "off the shelf" components at a price he can afford.

5.3 THE MICROPROGRAM CONTROLLER CIRCUIT

Let us now take a look at another LSI microcircuit. Another important circuit which has recently become available is the microprogram sequencer. These circuits are designed to serve as the next address control of the microprogram memory. They incorporate most the circuitry we discussed in section 2.2. Essentially every bipolar semiconductor manufacturer has a circuit of this type.

TABLE 2

Partial List of IBM 360/370 Instructions

		Type of Second Data Operand*							
	Operation	REG		HW	FW	MUL	IMD	CHR	DEC
Fixed	LOAD	LR		LH	L	LM	LA	IC#	
Point		LTR	LCR						
Arithmetic		LPR	LNR						
and	STORE			STH	ST	STM		STC#	
Logical	ADD	AR		AH	A				AP#
		ALR#			AL#				ZAP#
	SUBTRACT	SR		SH	S				SP#
		SLR#			SL#				
	COMPARE	CR		CH	C				CP#
		CLR#			CL#		CLI#	CLC#	
	MULTIPLY	MR		MH	M				MP#
	DIVIDE	DR			D				DP#
	AND	NR			N		NI#	NC#	
	OR	OR			O		OI#	OC#	
	EX-OR	XR			X		XI#	XC#	
	SHIFT	SRA	SRDA						
		SLA	SLDA						
		SRL	SRDL						
		SLL	SLDL						
Branching		BALR			BAL				
		BCTR			BCT				
		BCR			BC				
					BXH				
					BXLE				
Floating	LOAD	LER′	LDR"		LE′	LD"			
Point		LTER′	LTDR"						
		LCER′	LCDR"						
		LPER′	LPDR"						
		LNER′	LNDR"						
	STORE				STE′	STD"			
	ADD	AER′	ADR"		AE′	AD"			
		AUR′	AWR"		AU′	AW"			
	SUBTRACT	SER′	SDR"		SE′	SD"			
		SUR′	SWR"		SU′	SW"			
	COMPARE	CER′	CDR"		CE′	CD"			
	MULTIPLY	MER′	MDR"		ME′	MD"			
	DIVIDE	DER′	DDR"		DE′	DD"			
	HALF	HER′	HDR"						

*Type of Second Data Operand:

REG	Register (4 byte)
HW	Half Word (2 Byte)
FW	Full Word (4 Byte)
MUL	Multiple Words (4 or more Bytes)
IMD	Immediate (operand from instruction word)
CHR	Character (1 Byte)
DEC	Decimal

# Not implemented in 168/E
′ Implemented with optional Floating Point Processor
" Implemented with optional Floating Point Processor as REAL*6 rather than REAL*4

It is interesting to study one of them in detail in order to both have a basic understanding of their features and to illustrate some of the techniques used in microprogramming. A list of the currently available microsequencers is shown in table 3. Unlike the case of the microprocessor slice, none of these circuits seems to have taken a clear lead in popularity. As an example of a microsequencer to study, I have taken the newest and probably the most interesting one: the 291Ø.

Figure 39 is a block diagram of the AM291Ø made by Advanced Micro Devices. From this figure one recognizes the same basic structure that we used in the figure 27; i.e., the NEXT ADDRESS MULTIPLEXER, the incrementer, the MICROPROGRAM COUNTER register, and the condition code input (CC). To the basic structure of figure 27 some additional features have been added in order to make the circuit of more general utility. The NEXT ADDRESS MULTIPLEXER, for example has two additional inputs: one from a 5 word last-in first-out program counter STACK and the other from a register which can also be used as a counter.

The D input to the multiplexer come directly from the data inputs pins on the chip. They are intended to be used for both the BRANCH ADDRESS field of the PIPELINE REGISTER and the output of the MAPPING ROM as is shown in figure 4Ø. In order to chose between these two possibilities the 291Ø provides two outputs, PL and MAP which enable the outputs of the PIPELINE REGISTER or the MAPPING ROM respectfully. An additional OUTPUT-ENABLE signal is available for another register and/or ROM called VECT. The 291Ø will generate a signal on only one of the OUTPUT-ENABLE (OE) signals at a time, thus the three sources for the D inputs are effectively multiplexed. In the 291Ø the NEXT ADDRESS MULTIPLEXER is really a six input circuit, with 3 internal and 3 external sources.

Let us consider each of the NEXT ADDRESS MULTIPLEXER's inputs. First the MICROPROGRAM COUNTER input is identical to the one we studied in figure 27. The Carry-In (CI) should be set to '1' to make the incrementer add one to the current microprogram address.

A new feature is the R REGISTER/COUNTER. It has several uses which illustrate some of the other techniques one can use with the microprogramming. As a register, it is an auxiliary storage location to the BRANCH ADDRESS field of the program memory PIPELINE REGISTER. It is be loaded from the Direct

TABLE 3

List of Available Microprogram Sequencers.

Part no.	Originated by	Second Source	(bits)	Tech.	Comments
2909	Advanced Micro Devices.	Raytheon, National.	4	S-TTL	Slice
2911	Advanced Micro Devices.	Raytheon, National.	4	S-TTL	Slice, similar to above with additional features
2910	Advanced Micro Devices.	none	12	S-TTL	With Condition Code logic
3001	Intel	Signetics	9	S-TTL	Complex but saves memory space
67110	Monolithic Memories	none	9	S-TTL	With ALU shift matrix
74S482	Texas Inst.	none	4	S-TTL	Slice, simple
8X02	Signetics.	none	10	S-TTL	Very simple
9406	Fairchild	none	4	S-TTL	Slice
9408	Fairchild	none	10	IIL	Condition Code Register

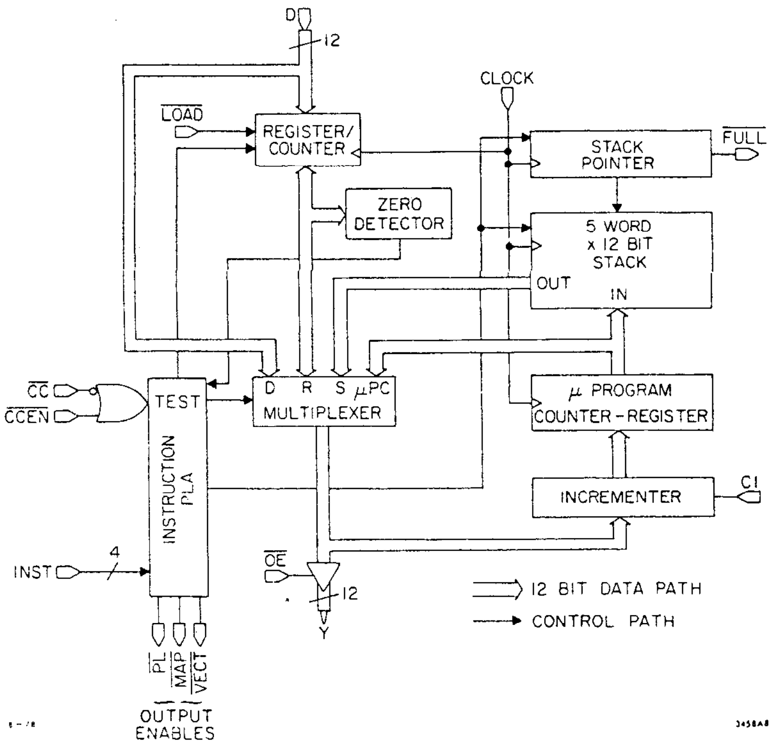


Figure 39: Block Diagram of 2910 Microsequencer

Data (D) inputs whenever the LOAD signal is received. At a later time in the program one could execute a conditional two way branch or subroutine CALL to either the D input or the R REGISTER.

The R REGISTER can also be used as a counter. This allows one to repeat an instruction or a series of instructions in the following way. The counter is initially loaded with a value. During certain microinstructions the counter is decremented by one. When the value of the counter reaches zero, a ZERO DETECT signal is generated. Other microinstructions can use this signal in place or in conjunction with the CC input of the circuit in order to select the next address. Thus when the counter has been decremented to zero one can have an instruction take an alternative branch to the normal one. This structure may be used for example in certain iterative instructions such as multiplication and division.

As with ordinary programming, there are advantages in using subroutines for certain sections of the program so that they need not repeated in the memory as many times as they are used. In order to make a subroutine CALL one needs to add two capabilities to the hardware structure we have already studied. First, when we make a BRANCH to the subroutine, we must have the capability of storing the address to which we should return after the subroutine execution is completed, and second, we must be able to return to that stored address.

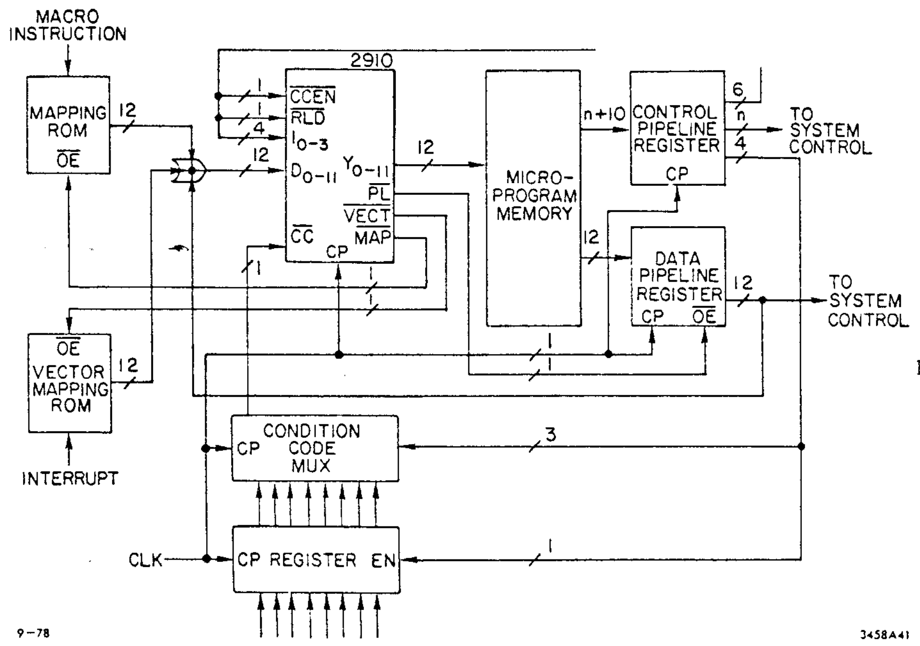


Figure 40: Typical Control Unit with 2910 Microsequencer

The 5 word STACK in the 2910 with its connection to the MICROPROGRAM COUNTER and to the NEXT ADDRESS MULTIPLEXER gives us both capabilities. It is used in conjunction with the STACK POINTER which is a up/down counter that always points to the last data entered into the STACK file. When the counter is incremented it is called a PUSH and conversely when it is decremented it is called a POP. A subroutine CALL is executed in the following way. The subroutine address is selected as the next microprogram address from either the D input or the R REGISTER. The MICROPROGRAM COUNTER will thus be address of the subroutine CALL plus 1. The STACK POINTER is first PUSHed and then the MICROPROGRAM COUNTER is stored into the top of the STACK. The next cycle will be from the first location of the subroutine. The subroutine RETURN is executed by selecting with the NEXT ADDRESS MULTIPLEXER the output of the STACK. Thus the next instruction to be executed will be one instruction beyond the instruction which made the subroutine CALL. At the end of the RETURN cycle the STACK is POPed to complete the linkage. Since the STACK contains 5 words, the subroutine CALLs can go to 5 deep; beyond that the highest level subroutine return address will be lost.

The circuit has a number of parts which need to be controlled. The output of the NEXT ADDRESS MULTIPLEXER must be selected from one of the four inputs or forced to zero; the STACK must be PUSHed, POPed, HELD, or ZEROed; the R REGISTER must be LOADed, DECRemented, or HELD; and one of the OUTPUT-ENABLES may generated. Of all the combinations possible, the manufacturer has selected 16 and he provides a four bit input so the user can provide a binary code for which possibility he wants. These four bits are called the 'microprogram controller instruction'. In most applications, four bits of output from the microprogram memory are used to provide this instruction in the same way that bits 4 and 5 were used in the simple scanner example. Some of the instructions use the CC input for conditional branching. In When the CC input is 'true' it is called the PASS state and when it is 'false', it is called the FAIL state. In addition the CONDITION CODE ENABLE (CCEN) input can be used to force the internal condition code (TEST) to PASS. Finally, the LOAD input of the R REGISTER can be independently controlled. Thus we have really 6 bits of instruction input, although some of the 64 combinations are redundant.

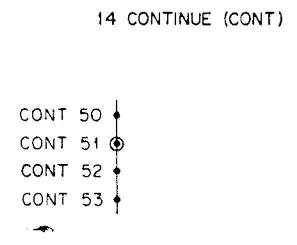
We will now go through all 16 of the 2910's microinstructions. This exercise will serve to illustrate the special techniques one can use in microprogramming. It is also rather interesting and fun. For each instruction we shall consider the state of the TEST input and the contents of the R REGISTER/COUNTER since they may alter the resultant operation of the microinstruction. When their states do not affect the operation, it is called a "Don't Care" condition which is indicated by an "X" in the figures that are to follow. The microinstruction may affect the contents or status of the STACK, next address source, the R REGISTER/COUNTER, and/or the OUTPUT-ENABLES. If the operation does not affect any one of them it is called a "No Change" condition which is indicated by "NC" in the figures. As we study the 2910's microinstructions, one can try to imagine an analogy with FORTRAN statements that control the program flow.

5.3.1 Continue.

Instruction 14 is a Continue (CONT) which is the simplest instruction. The next address source is always the contents of the MICROPROGRAM COUNTER. One should recall that the MICROPROGRAM COUNTER is always the current address output of the 2910 plus one. As show in figure 41, the status of the TEST input and R REGISTER don't influence the operation, the STACK and R REGISTER don't change their value and the PIPELINE REGISTER is enabled. The Continue instruction is probably the most frequently used instruction since it is used when a series of microinstructions are executed.

5.3.2 Jump Map.

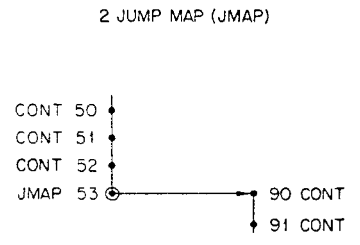
Instruction 2 is a unconditional branch instruction in which the Mapping ROM OUTPUT ENABLE is turned on. It is called a JUMP MAP (JMAP). The



TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
X	X	NC	PC	NC	PL

Figure 41: 2910 Instruction 14 .

status of the TEST input and the R REGISTER are Don't Care. The next address source is always taken from the D input. In the example given in figure 42, microinstruction 53 has the JMAP instruction. When it appears in the PIPELINE REGISTER, the MAPPING ROM is enabled and its output is routed through the 2910 to the address input of the microprogram memory. If the contents of the MAPPING ROM were 90, then the program flow would jump from 53 to 90 as shown. In FORTRAN the JMAP instruction is analogous to the GO TO statement.

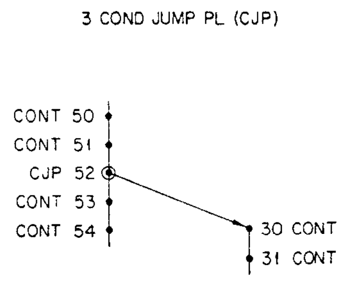


TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
X	X	NC	D	NC	MAP

Figure 42: 2910 Instruction 2 .

5.3.3 <u>Conditional Jump Pipeline</u>.

Instruction 3 is a conditional branch instruction in which the PIPELINE REGISTER OUTPUT-ENABLE is turned on. It is called Conditional Jump Pipeline (CJP) and it is illustrated in the example given in figure 43 . If the status of the TEST input is Fail, then the next address source is taken from the MICROPROGRAM COUNTER. So in the example, the program flow would be from instruction 52 to instruction 53. On the other hand, if the status of the TEST input is Pass, then the next address source is the D inputs. Thus the program flow in the example goes from instruction 52 to instruction 30. The contents of the R REGISTER are Don't Care and the STACK and R REGISTER are unaffected. One should recall that one can use the Condition Code Enable (CCEN) input to



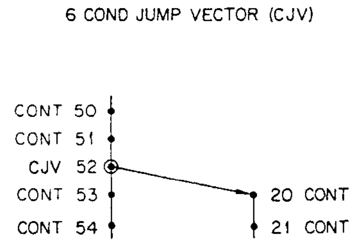
TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
PASS	X	NC	D	NC	PL
FAIL			PC		

Figure 43: 2910 Instruction 3 .

force the internal condition to Pass. Doing this changes the CJP instruction into a unconditional jump to the contents of the PIPELINE REGISTER. The CJP instruction corresponds to the FORTRAN statement "IF(...) GO TO".

5.3.4 <u>Conditional Jump Vector</u>.

An almost identical instruction is instruction 6 which is illustrated in figure 44 . The only difference is that the VECT OUTPUT-ENABLE is turned on instead of the PIPELINE OUTPUT-ENABLE.

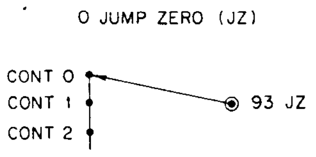


TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
PASS	X	NC	D	NC	VECT
FAIL			PC		

Figure 44: 2910 Instruction 6 .

5.3.5 <u>Jump Zero</u>.

A very special instruction is instruction 0. In this instruction the output of to 2910 is forced to a binary zero, thus it is called the Jump Zero (JZ) instruction. In the same instruction the STACK is cleared and the PIPELINE REGISTER is enabled. The intention behind this instruction is to put the microsequencer into a well defined state when the power is first turned on. It is easy for the user to add circuits so that on power up the microinstruction 0 is issued to the 2910. Figure 45 illustrates this instruction.

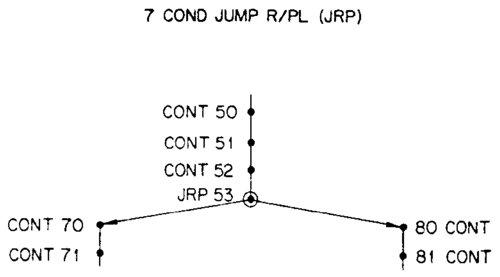


TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
X	X	CLEAR	0	NC	PL

Figure 45: 2910 Instruction 0 .

5.3.6 Conditional Jump R/PL.

Instruction 7 is the first example which uses the R REGISTER. It is a Conditional Jump R or PIPELINE REGISTER (JRP) and is illustrated in figure 46 . When the TEST input is PASS, the next address source is from the D inputs with the PIPELINE REGISTER enabled. When the TEST input is FAIL, the next address source is from the contents of the R REGISTER. One should recall that the R REGISTER may be loaded in any instruction by generating the LOAD signal. This instruction is effectively a two way Jump, since the next sequential address is never the next address source. In FORTRAN it would correspond to two statements: an "IF(...) GO TO" followed by "GO TO". In a microprogram with the 2910, the two way branch is only one instruction.

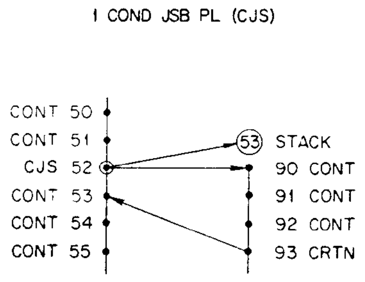


TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
PASS	X	NC	D	NC	PL
FAIL			R		

Figure 46: 2910 Instruction 7 .

5.3.7 Conditional Jump Subroutine Pipeline.

Subroutine CALLs can be made with instruction 1. As shown in figure 47, the Conditional Jump Subroutine (CJS) instruction is actually a conditional subroutine CALL. If the TEST input is FAIL, the next address source is the contents of the MICROPROGRAM COUNTER which is the next sequential instruction. If TEST input is PASS, then the next address source is the D input with the PIPELINE REGISTER enabled. The STACK COUNTER is PUSHed and the current contents of the MICROPROGRAM COUNTER are stored in the STACK, thus saving the address to which the subroutine return should be made. The CJS instruction can be modified to a unconditional subroutine jump by using the CCEN input to force the



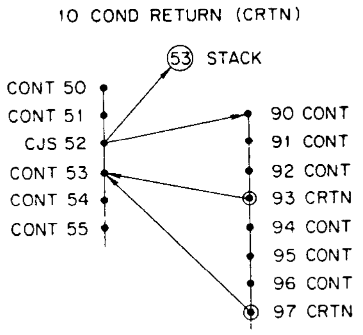
TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
PASS	X	PUSH	D	NC	PL
FAIL		NC	PC		

Figure 47: 2910 Instruction 1 .

TEST input to the PASS state. The FORTRAN equivalent of this microinstruction would be "IF(...) CALL".

5.3.8 Conditional Return.

The return from subroutine is executed by instruction 10. As shown in figure 48, it is also a conditional instruction. If the TEST input is FAIL, the next address source is taken from the MICROPROGRAM COUNTER with no other change. If the TEST input is PASS, then the next address source is the contents of the top of the STACK and at the end of the microcycle the STACK POINTER is POPed. Again this instruction can be modified to a unconditional return by using the CCEN input. The FORTRAN equivalent would be "IF(...) RETURN".

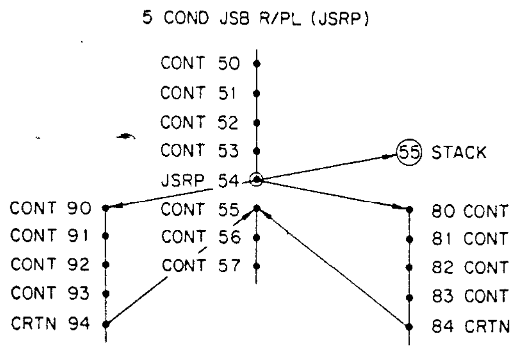


TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
PASS	X	POP	STACK	NC	PL
FAIL		NC	PC		

Figure 48: 2910 Instruction 10 .

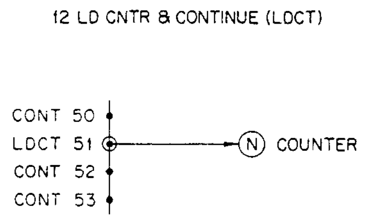
5.3.9 Conditional Jump Subroutine R/PL.

Another method for making subroutine CALLs is the Conditional Jump Subroutine Register/Pipeline (JSRP) as shown in figure 49 . If the TEST input is FAIL, then the next address source is taken from the contents of the R REGISTER. If the TEST input is



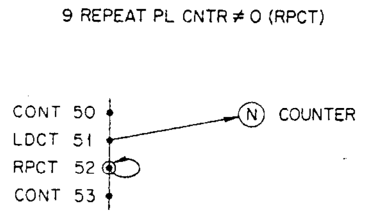
TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
PASS	X	PUSH	D	NC	PL
FAIL			R		

Figure 49: 291Ø Instruction 5 .



TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
X	X	NC	PC	LOAD	PL

Figure 5Ø: 291Ø Instruction 12 .



TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
X	=0	NC	PC	NC	PL
	≠0		D	DEC	

Figure 51: 291Ø Instruction 9 .

PASS, then the next address source is taken from the D inputs with the PIPELINE REGISTER enabled. In either case the STACK POINTER is PUSHed and the contents of the MICROPROGRAM COUNTER is stored at the top of the STACK. Thus the TEST input determines which subroutine is CALLed and not whether one CALLs a subroutine or not. The FORTRAN equivalent is somewhat more complex then the ones we have seen so far. It might be written as "IF(...) CALL X" followed by "IF(.NOT.(...)) CALL Y". In some other high level programming languages this micro-instruction might be expressed as a "IF(...) THEN CALL X ELSE CALL Y". Again we see that in the microprogram it is only one instruction.

5.3.1Ø Load Counter and Continue.

The "Load Counter and Continue" (LDCT) instruction provides an alternate method of loading the R REGISTER. As shown in figure 5Ø, the next address source is always the MICROPROGRAM COUNTER just like the Continue instruction. The R REGISTER is loaded from the D inputs with the PIPELINE REGISTER enabled. Many microprocessors could use this instruction as the only method of loading the R REGISTER thus eliminating the need to control separately the LOAD input to the 291Ø. In FORTRAN, this microinstruction might be equivalent to setting the end point of a DO LOOP as will be seen below.

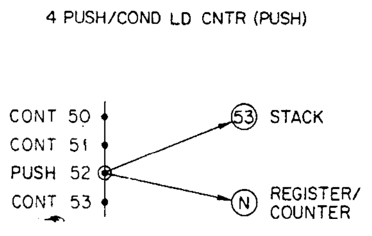
5.3.11 Repeat Pipeline Counter Not Equal to Zero

The next instruction is the first example of using the R REGISTER as a counter. It is called "Repeat Pipeline Counter Not Equal To Zero" (RPCT). If the contents of the R COUNTER are not equal to zero then the next address source is taken from the D inputs with the PIPELINE REGISTER enabled. At the end of the cycle, the R COUNTER is also decremented by one. If the contents of the R COUNTER is zero, then the next address source is taken from the MICROPROGRAM COUNTER and the R COUNTER is left unchanged. As illustrated in figure 51, the RPCT instruction can be used to force execution of the same micro-instruction many times by letting the contents of the PIPELINE REGISTER be equal to the address of the instruction. This may be used, for example, to do iterative multiplication or division micro-instructions. The combination of 291Ø micro-instructions 12 and 9 look very much like the FORTRAN statements

```
    DO 1Ø I=1,N
    (one or more statements)
 1Ø CONTINUE
```

5.3.12 Push/Conditional Load Counter.

Another instruction which loads the R REGISTER/COUNTER is show in figure 52 . It is in fact a conditional load of the counter and it is called "Push and Conditional Load Counter" (PUSH). If the TEST input is FAIL, then the R REGISTER/COUNTER is not loaded while if it is PASS then it is loaded from the D inputs with the PIPELINE REGISTER enabled. In either case the next address source is from the MICROPROGRAM COUNTER, the STACK COUNTER is PUSHed and the MICROPROGRAM COUNTER is stored at the top of the STACK. The purpose of this instruction will not be clear until we study the next and last 4 micro-instructions.



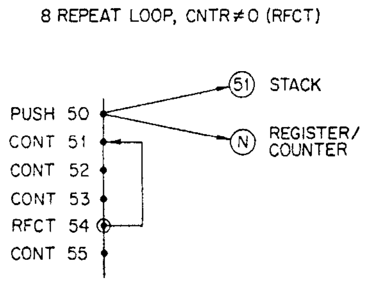
TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
PASS	X	PUSH	PC	LOAD	PL
FAIL				NC	

Figure 52: 291Ø Instruction 4 .

5.3.13 Repeat Loop, Counter Not Equal Ø.

The next instruction works with the PUSH to perform a microprogram DO-LOOP as is shown in figure 53 . It is called "Repeat Loop for Counter not equal to Zero" (RFCT). The instruction is a conditional jump using the contents of the R REGISTER/COUNTER as the TEST input. If the contents are not equal to zero, then the next address source is taken from the top of the STACK and the R COUNTER is decremented. In other words the program branches back to the beginning of the loop. When the contents of the COUNTER becomes zero, then the next address source is taken from the MICROPROGRAM COUNTER and the STACK is POPed while the COUNTER is left unchanged. In other words the program drops through the bottom of the loop. Thus we see that the R REGISTER/COUNTER is used like the running index of the DO LOOP. The STACK is used in this case to save the beginning of the LOOP rather then for saving the subroutine return address. In fact the STACK can be used as a combination of both up to 5 levels of loops and subroutines. The combination of the PUSH and RFCT microinstructions looks very much like the FORTRAN statements:

```
      DO 1Ø I=1,N
      (one or more statements)
   1Ø CONTINUE
```

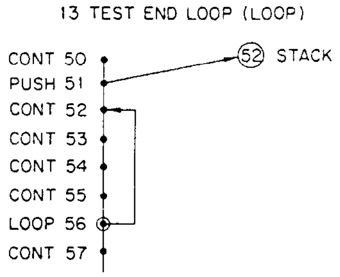


TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
X	=0	POP	PC	NC	PL
	≠0	NC	STACK	DEC	

Figure 53: 291Ø Instruction 8 .

5.3.14 Test End of Loop.

Another example of looping is an instruction called "Test End of Loop" (LOOP). It operates the same way as the RFCT instruction except that the Condition Code input is used as the TEST input rather than the contents of the R REGISTER/COUNTER and the counter is not affected. Note that in the example shown in figure 54 if one never got a TEST input PASS status one would have an infinite loop. Note also that although the PUSH instruction was used at instruction 51 in order to save the beginning address of the loop, the R REGISTER/COUNTER is not used in the loop. In FORTRAN, the LOOP microinstruction looks like a simple "IF(...) GO TO".



TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
PASS	X	POP	PC	NC	PL
FAIL		NC	STACK		

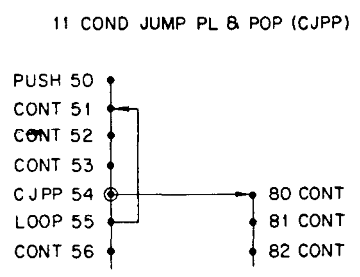
Figure 54: 291Ø Instruction 13 .

5.3.15 Conditional Jump PL and POP.

Each PUSH of the STACK must be followed somewhere by a POP in order to not to lose the subroutine linkage. Instruction 11 has been designed to enable one to conditionally jump out of a loop and restore the STACK at the same time. It is called the "Conditional Jump Pipeline and POP" (CJPP) and it is illustrated in figure 55 . If the TEST input is FAIL the next address source is the MICROPROGRAM COUNTER and the STACK is left unchanged. If the TEST input is PASS, then the next address source is taken from the D input with the PIPELINE REGISTER enabled and at the same time the STACK is POPed.

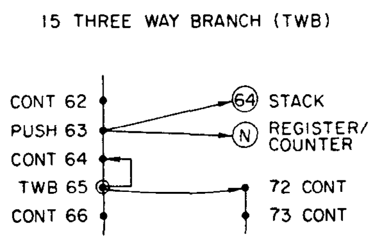
5.3.16 Three-Way Branch

The next and last instruction is the most complex of all. It uses both the TEST input and the contents of the R COUNTER to determine one of three next address sources. It is appropriately called "Three Way Branch" (TWB). It is also used with the PUSH instruction as shown in figure 56 . As long as the TEST input is FAIL, the instruction operates like the RFCT, that is, it the microprogram branches back to the address contained at the top of the STACK as long as the R COUNTER is non-zero. When the R COUNTER reaches zero, however, the next address source is taken from the D inputs with the PIPELINE REGISTER enabled. If the TEST input is PASS, then the program drops out of the loop by taking the next address source from the MICROPROGRAM COUNTER and the STACK POINTER is POPed. In this case the R COUNTER is decremented or unchanged depending on its value.



TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
PASS	X	POP	D	NC	PL
FAIL		NC	PC		

Figure 55: 2910 Instruction 11 .



TEST	REG/CNTR DATA	STACK	ADDRESS SOURCE	REG/CNTR	$\overline{OE}$
PASS	= 0	POP	PC	NC	PL
	≠ 0			DEC	
FAIL	≠ 0	NC	STACK	DEC	PL
	= 0	POP	D	NC	

Figure 56: 2910 Instruction 15 .

This strange instruction turns out to be quite useful. If in a loop one were searching for a data point in memory, for example, then the loop could end when either the data point is found (TEST input becomes PASS) or by reaching a certain limit (R COUNTER becomes zero). Note that the in the two ending conditions the program goes to two different locations. Thus when compared to a FORTRAN program, this instruction is like having an "IF(...)GO TO" statement as the last statement in a DO-LOOP.

5.3.17 Summary of 2910.

This completes the study of the microinstruction of the 2910. To the FORTRAN programmer these instructions should not seem too strange at all. There is a big difference, however, in the manner in which the instructions are executed. With a FORTRAN program running on a normal computer the compiler has generated various machine instructions to get the desired program flow. With the microprogram sequencer, the program flow is controlled within one microinstruction. Hence we see that microsequencers are designed to make microprogram fast and efficient is memory space by minimizing the number of instruction steps to control the program flow. One must remember that besides the PIPELINE REGISTER bits which control the microsequencer, there other bits which control that which is being controlled. The sequencer does not do useful data manipulation itself.

REFERENCES

1. [1] Martyn Edwards and Erik Dagless, Microprocessors 1, 407 (1977).

2. [2] Phillip M. Adams, SIGMICRO Newsletter vol 9 no 1, 23(1978); and vol 9 no 2, 7(1978).