THE DESIGN OF COMPUTERS USING BIT-SLICE TECHNOLOGY

Michael E. Valdez, Ph.D., P.E.

Department of Electrical and Computer Engineering

Florida Institute of Technology

Melbourne, Florida

COURSE CONTENT

Florida Institute of Technology

Department of Electrical and Computer Engineering

CP 4277 - Introduction to Large Computer Design

Course Outline

Course Content

1. Introduction. Purpose of the Course. Tests. Grades. Reading.
2. Computer Architecture. Elements. Characteristics. Implementations. The microcomputer. The bit-slice concept.
3. The Control Unit. Functions. Implementations. Hard wired logic or PLA. Microprogrammed logic.
4. The Microprogram. The Nanocomputer. Microprogram controller. Timing. Pipelining. Microprogramming.
5. The Arithmetic Logic Unit. Architecture. Implementations. Timing. Microprogramming.
6. Speeding up the Arithmetic Logic Unit. Carry Look Ahead. Multiplication. Other Functions.
7. The Processor Status. Bit Slice Solution. Signals. Microprogramming. The Micro Status Register. Uses.
8. The Program Controller. Functions. Implementations. Discrete Logic. Bit Slice. Microprogramming.
9. Brief Review of Other Bit Slice Devices. Characteristics. Uses. Trends in Industry.

GOALS:

To give the students an introduction to the design of large computer systems and special purpose controllers. To give the students the theory behind concepts like bit-slice, microcode, pipelines, etc. To develop a certain expertice in the design of systems using bit-slice and microcode.

Textbook:

Bipolar Microprocessor Logic and Interface Data Book, Advanced Micro Devices, lastest edition.

Reference:

Bit Slice Microprocessor Design, Mick and Brick, Mc Graw Hill Book Co. 1980.
TTL Manual.

Pre-requisites:
1.- Computer design using microprocessors.
2.- The design of computer interfaces.

Estimated ABET Category:
Engineering Theory: 1 credit
Engineering Design: 2 credits.

THE DESIGN OF COMPUTERS USING BIT-SLICE TECHNOLOGY

Michael E. Valdez, Ph.D., P.E.

Slices: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9

INTRODUCTION

This course is devoted to the design of large computers and advanced controllers, mainly using bit-slice technology. The course starts with an analysis of preliminary subjects and continues developing the details of the design, through the solution of examples, that are carried through the course. An important by-product of this course, for graduate students, is to give them ideas for research.

The classes are developed as a sequence of examples. During the solution of these examples, and motivated by the need to use them, the different large scale integration devices available for bit-slice design are studied and their characteristics analyzed. This permits the students to determine if their use in the design on hand is justified or not, and to choose between them.

The same procedure is used for the description of advanced techniques for computer design. Methods to speed up the operation, pipelining, cache memory, etc. are topics that are discussed as possible alternatives for the design of the computer and the controller and, in this context, are easier to understand and to use, rather than in a cold recitation of qualities and problems.

In the topic of advanced computers and architecture, an analysis is made of the design of a stack computer and high level instructions. As the course develops, many advanced topics are analyzed, discussed, and their range of application is evaluated.

It is important to note that the emphasis of any course in computer design must not be on the details of design or the use of particular devices but on the theory of design, on general purpose procedures that can be used with the current devices or with any other device that might be developed later, to study alternatives of design so, when the student graduates and go out into the world, he/she is prepared for the future, not for the present.

At this moment all the students know how to design the memory circuits and the input and output circuits; the connection of the processing unit is trivial, similar to the connection of a port. The only point the students have not yet seen is the design of the control unit. The main emphasis of this course is then, in the design of the control unit, whether for a controller or for a computer.

This text is divided in slices (naturally) each covering a topic. Because of this, some slices are longer than others and the slice is not a class period. It is expected that the students will obtain as a minimum a copy of the AMD Bipolar Microprocesor Logic and Interfaces data book, as a reference for the solution of the projects. This text makes extensive reference to the values in the data book. A normal TTL data book is also required.

The Appendix was written by Jerald W. English who, during the Spring of 1985 designed and built the Bit Slice Trainer he describes, as part of a Special Project in Computer Engineering. The trainer is available for students use in the Computer Research Laboratory.

THE DESIGN OF COMPUTERS USING BIT-SLICE TECHNOLOGY

Michael E. Valdez, Ph.D., P.E.

SLICE #1

Slices: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9

COMPUTER ARCHITECTURE

This Slice constitutes an introduction to the topics covered in this course. Since this course is devoted to the design of computers using bit-slice technology, the definition, classification and elements of a computer are studied in this Slice, to develop common terminology. This Slice finishes with an analysis of the procedures we use to design computers.

1.1.- **Computer Generations**.

The purpose of this course is to design computers, large computers, so might as well we start by looking at what is a computer. The computer originated during the second world war, developed as a calculating machine. Today, more than 40 years after the first computer was developed, most of the computers are used as calculating machines. Throughout these 40 years of the life of computers we have had several changes in technology and characteristics of the computers but, in spite of these changes in technology, the advances in programming sciences, etc. all computers built in this country have the same architecture. There is only one architecture of computers up to this time. This architecture, first studied by von Newmann, has the program stored in memory and it works sequentially, one instruction at a time.

We can classify the computers in very different ways and we might be interested to do it before we face the design. The first generation of computers is considered to be the one built with vacuum tubes, very simple, very hot, very large. The capabilities of these large first computers was very limited. There is a story, popularized by Radio Shack, that the SSEC computer (Selective Sequential Electronic Calculator) was used by astronomer Walter J. Eckert to compute the ephemeris for the planet Jupiter and the external

planets, taking 120 hours to do the job. A man will need in the order of 80 years to do the same job. A TRS-80 model I will do the same job in 11 hours and more modern personal computers will do it in under one hour.

The first breakthrough in technology were the transistors. The transistor reduced the size and the heat dissipation of the computers. The only difference between the first and second generation of computers was that the switching device was the transistor instead of the vacuum tube.

The third generation was brought by the integrated circuits. The transistor are solid state devices and it was discovered that they can be made smaller and smaller and can be put in packages to perform complex functions. This is the way the minicomputers appear. Computers that were smaller than the predecessors. There was no other difference between the first, second and third generation of computers.

As the integrated circuit technology advanced, the capabilities of the computers increased. In the time of the transistors, a computer with 16 K of memory was a big computer. Memory was all magnetic cores. With the advent of integrated circuits, solid state memory was practical, using flip flops as storage device, and computers with large memory space started to appear.

The fourth generation of computers was brought by the use of very large scale integration. Most of the devices that perform the functions of the computer are integrated into one or a few integrated circuits. Again, there is no substantial difference between computers from the first, second, third and fourth generation, other than size, heat, power consumption, etc. There is no change in architecture.

The Japanese came a few years ago with the term "fifth generation" computers to represent advanced computers with more capabilities and speed than the current computers. This does not represent a technological breakthrough like the others that have been used to mark the different generations of computers. This does not represent either a change in the architecture. More than anything, the Japanese advocates a computer devoted to non-numeric computations, with advances specially in software. This computer will be called by us an intelligent computer but it does not represent a departure from the current type of computer. A real fifth generation computer would be a change in technology, like using optical switching devices instead of transistors.

1.2.- **Computer Classification**.

There is another way we can look at computers. Recall that all computers have the same architecture, the same structure. At the beginning all the computers were the same, big, hot, limited capabilities, so there was no problem. There was different names given to the different machines, but they all were computers. Now we have quite a variety of computers from the portable personal computer you can carry in your pocket to the large computers used by the military, that fill several racks. There are different names given to these computers creating large confusion. A manufacturer might call one of his products at the same time a microcomputer when he wants to emphasize its size or the fact that it is a single integrated circuit, or a super-minicomputer when he wants to emphasize its capabilities. We use these names with perfectly

defined meanings as follows:

We call *microcomputers* those computers that are very simple, have very few resources and signals, used for single user, single task; the units are so cheap that it does not make sense to multiplex them; the speed is low and we measure it as the number of floating point additions per second; a microcomputer can perform in the order of 1000 flops. The instruction set is special, each instruction is specific for the particular resource. The input and output is done by ports; that is, we have a unit that is part of the computer that permits us to connect any peripheral we want to connect to the computer. Examples are the 6502, the 6811, and similar units.

We use the name of *minicomputer* to represent a more complex computer, with more resources, more signals, with limited capabilities for multitasking, the speed is about ten times higher, the instructions become primitive because the computer has more resources and we cannot devise one instruction for each operation with each resource; the input and output is also with ports. Examples are the 68000, the Z80, even the 8086.

The *main frame* has more complexity because it has more resources, more signals; the purpose of the main frame is to do multitasking for many users. This does not mean that we cannot have a main frame performing a single task. Typical example is the IBM 370. The speed is about ten times higher, in the order of 100 KF, the instruction set is primitive, the input and output devices are connected directly to the bus. The reason is that this device is not mass produced and, in general, it is not possible to connect any peripheral to the computer. The manufacturer normally connects the peripherals. The VAX is some place in between the minicomputer and the main frame; it is called a super-minicomputer.

Then, we have the *super computer* with many more resources and signals. A super computer is a special purpose computer, it is designed for a certain task; the speed is at least 10 MF, the instructions are again special since there is no general purpose super computer. The input and output is connected to the bus. An example of super computer is the CRAY.

Finally, the *non-numeric computer*: the resources and signals are very complex, this computer can also be called intelligent computer. The use is special because the few ones that exist have been developed for a special project. In this country they are called LISP computers, in other countries they receive other names. The speed of these computers cannot be compared with the others because, although they can perform numerical operations, their main function is to perform non-numerical functions. The instruction set naturally is special and the input and output is connected to the bus.

Now, it is interesting to see where we are, and what we will be doing in this course. We will be talking about the design of a computer where we will design everything, including the instruction set, normally starting from the functions to be performed. We will be then talking about minicomputers as a minimum, but more logically about main frames.

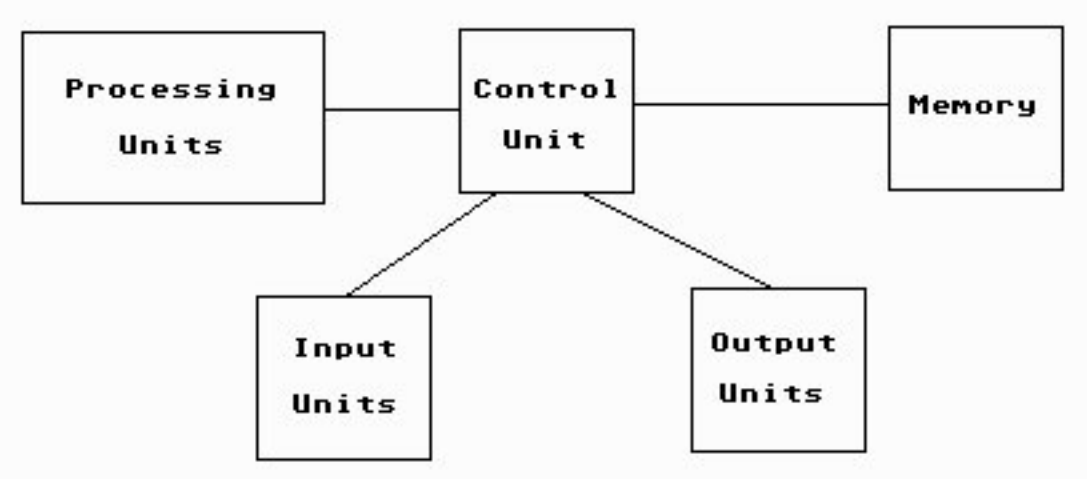
An interesting comparison of several computers is done in the following table where the cost and speed

of widely different computers is tabulated with the resulting cost per megaflop in the last column.

Comparison between Computers			
Computer	Cost K$	Speed Mflop	M$/Mflop
Atari 1040 ST	1	.01	.01
IBM PC, no math coprocessor	4	.00037	10.8
IBM PC, with coprocessor	5	.0073	.68
VAX 11/750	80	.057	1.4
DEC KL-10	480	.18	2.7
CDC 7600	4500	4.6	.98
Cray 1S	11000	18.0	.61
Cray XMP	15000	53.0	.28

1.3.- **Computer Elements**

The easiest way to look at the computer elements is to look at a block diagram of a very simplified computer. In this block diagram, like Fig 1.1, we find five basic elements of the computer: We have memory where the program and the data are stored, we have an element that permits us to enter information into the computer, we have an element that permits the computer to send information to the outside world, we have an element that permits the computer to manipulate and transform the information, and finally, an element that controls the operation of all the other elements. There can be more than one of each unit, with the exception of the control unit, that is unique.



These elements have well defined names that we will use throughout the text. The first element, where

we store the program and the data, is called the memory; the second element, that permit us to input data into the computer, is called the input unit; the third element, that permits the computer to send data to the outside, is called the output unit; the element that permits manipulation and/or modification of the data is called the processing unit; and, finally the element that controls the operation of the computer is called the control unit.

1.4.- **The Procedure for Design**.

Designing a computer, or a specialized controller is not a trivial operation. It requires experience, intelligence and good engineering judgement. The design naturally starts with the definition of the function or functions to be performed by the unit, its requirements and limitations, its constraints, etc. From this information, the unit is catalogued as a controller, a computer controller, or a computer. This classification might have to be changed later when the design has advanced. This textbook will be concerned with the design of computers and computer controllers.

Whether a computer or a computer controller, the functions and characteristics are transformed into instructions. In the case of a computer, this leads to an instruction set. In the case of a controller, the idea of instruction is not always clear. The elementary functions that the controller executes can be considered instructions. When the controller works with a computer, it receives instructions, commands and requests from the computer; these can be analyzed as the instructions in a computer.

The design of the unit can be done in several ways and no organized method for design has yet evolved in industry. Most of the design is done intuitively today. It is time for this to change. This textbook presents a step by step procedure that, not only results in the proper design of the unit, but that has as a by-product that the microcode is produced without effort. The procedure presented in this textbook is based on the analysis of the instructions one by one, first from the point of view of the steps required for their execution, defining simultaneously the hardware and the microcode. Later, a timing analysis is done following the same procedure of instruction by instruction, which produces not only the minimum clock period, but identifies the bottlenecks, the places where improvement is possible, the places where the unit can be speed up by overlapping steps, etc.

1.5.- **Computer Design**

When talking about computer design it is necessary to distinguish between three expression that are very close together, so close that many people confuses them. These three expressions for the idea of computer design are:

- Designing with a computer as a tool;
- Designing with a computer as a part;
- Designing the computer itself.

It should be clear that the term computer design is used for these three different ideas, and this is bad.

Designing with a computer as a tool is called computer design, but it really is computer aided design, and this is the proper term to use. This course does not cover computer aided design, which is the subject of other courses.

When the term computer design is used to mean designing with a computer as a part, what we will be doing is designing a circuit or a system, of whatever complexity, but where one or several of the integrated circuits we will use, will be computers or controllers. In this case, we will need to follow the specifications of the manufacturer of the computer. By this we mean, we will need to learn the instruction set of the computer, the signals the computer puts out and requires from the system, the protocols for the different operations like memory accesses, interrupt handling, etc. We will also have to understand the timing constraints of the computer and design our system in such a way that these constraints are satisfied.

Finally, designing the computer means that the designer will design every element of the unit; he does not have any rules or specifications to follow. The instruction set, the signals, the protocols, etc. have to be determined by the designer and he can do this without any restriction or constraint, except for his own design. The designer must select between many alternatives, many of them conflicting, based only on his own analyses and using his own criteria.

The purpose of this course is to give the students an understanding of the principles of computer design, the problems the designer faces, the tools he has to obtain results, the extent of his freedom, the extent of his responsibilities, and how he should go about to get the desired results.

The designer starts with an idea. The final result is a circuit diagram with devise selection, an instruction set, a timing analysis, a protocol of operation.

The designer starts with an idea of what he wants to achieve, normally without knowing how that can be achieved. This idea can be in the form of performance, characteristics, main field of use, etc.

Probably the first decision the designer makes is the bus size. Address space, data bus size, fields (program and data memory, system and user memories, single or multiuser, computation intensive, I/O intensive, etc.)

The next decision will be the technology: VLSI, bit-slice, discrete logic, custom chips, will be selected normally based on previous experience and the decisions already made.or its execution are developed. These steps will indicate if additional hardware is needed and the microcode signals required.

At this moment, several important decisions are made: if the additional hardware needed for the execution of an instruction seems excessive and that it will not be used for other instructions, the instruction could be deleted from the set, it might be modified, it might be replaced by similar instructions, or it might be accepted as it is.

The modification of one instruction might indicate the need to modify other instructions, whether because the added hardware permits to simplify their operation, or because the new instruction replaces the old one. Whether the procedure starts with simple instructions towards the complex ones, or from the complex to the simple, this possibility is always present.

The analysis of the instructions, breaking down them into steps, analyzing the additional hardware needed and the signals from the microcode, produce the full circuit diagram, the full microcode, and a workable instruction set.

The next step is the timing analysis of the instructions and their optimization.

Each line of the microcode is analyzed for the time it requires, considering all the parallel paths through which signals must travel. The final time for each line is not necessarily the longest of all parallel paths. Optimization can be applied here by combining several separate short steps into one, by producing the proper signals at the proper time; or breaking a long step into several, to permit the other steps to run faster.

Another possibility is the variation of the clock cycle in a line by line basis. It should be noted however, that there is a 6 to 11 nanoseconds penalty paid in the internal delay in the clock chip, for each change in clock frequency. Excessive use of this facility could then produce results that reduce the speed of the computer.

The timing analysis is not the last step of the design but another step that will suggest changes in hardware, changes in the instruction set, or even changes in the basic ideas of what the computer is supposed to do.

The purpose of this curse is then to introduce you to this problem and, by the way of examples, give you an idea of the methods for the solution. There is a very large number of details that will not be mentioned, because they belong to more advanced courses.

THE DESIGN OF COMPUTERS USING BIT-SLICE TECHNOLOGY

Michael E. Valdez, Ph.D., P.E.

SLICE # 2

Slices: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9

THE CONTROL UNIT

You are computer engineers, or you are trying to be computer engineers and, by now, you should know what is a computer engineer. A computer engineer designs computers so, we should study what is designing a computer. In other courses you have studied how to add memory to a computer, how to design interfaces, etc. but this is not designing a computer. In all the cases, an integrated circuit has been used as the control unit, the so called Central Processing Unit or CPU. Designing a computer is designing the control unit. Designing the control unit means defining the characteristics of the computer, its architecture, its instruction set, its capabilities, etc.

Designing a computer is really the fun part of being a computer engineer; probably there are very few jobs a computer engineer can have that compare to the satisfaction produced by designing a computer.

2.1.- **Control Unit Elements**.

In order to design the control unit of a computer we have to know what is the control unit, what it does. First, let us start by analyzing what parts the control unit has to perform its functions. Any way we analyze it we get to the same elements. The control unit has to have memory to remember the instruction because the fundamental thing that the control unit does is to interpret the instruction. The control unit has to have some way of getting data from memory and sending data to memory; consequently it needs some way to generate the address of the memory location it must access. It needs to generate the signals

that will indicate to memory if it is a read or write operation. The control unit has to have the capability of performing all the functions that our computer performs.

Now, what do we need?; we need first a program counter. Recall that we are talking only about sequential computers. When we design the control unit we will depart from this and then you will have a very good insight of what is a sequential computer. Normally we design the control unit as a non-sequential computer, or as a semi-sequential computer.

So, we need a program counter to keep track of where in memory is the program we are executing. We need some way to update and change the program counter, meaning that we need a way to increment the program counter; depending on the architecture we develop we increment by one, increment by two, or whatever number we need; but this is not enough, the program counter cannot be a counter that we send a clock when we want to increment it; we need ways for changing the program counter, ways to initialize the program counter, ways for getting the program counter and adding something like a part of the instruction for the relative branches, and put it back into the program counter; we need capabilities for doing arithmetic operations with the program counter.

We need a register in which we store the instructions. Normally we call this register the instruction register. The instruction register is one of the interesting parts in which we depart from the sequential operation of the machine. Depending on how we develop our control unit, it is possible we need the opcode of the instruction only for a moment; consequently, we can gain speed in the operation of our machine by getting the next instruction opcode into the instruction register before the end of the previous instruction, in what is called pre-fetching. In this way, when the instruction is executed, the next instruction is ready to start execution as soon as possible. This general procedure is used very much in computer design and speeds up very much the operation of a computer.

If our unit is synchronous, we need some way to synchronize all the parts, a clock. As we mentioned before, this clock is not necessarily constant frequency, because we can change the frequency, we can change the period every period. At the beginning of the period the control unit can tell the clock generator how long the period should last. This is one of the most interesting ways for speeding up a computer. Each clock lasts only as much as needed.

Finally, we need logic to develop all the signals not only for outside the control unit but for the inside elements of the control unit. So, there we go again, as we mention briefly before, this logic is another computer. This is similar to one of those drawings where there are two mirrors and you see one them reflected in the other, to infinity. It is evident that we cannot go to infinity. We cannot develop the logic as another computer because this logic comes out exactly the same way we got to the control unit. So we have computer inside computer, inside computer, to infinity. There are two very important reasons why we cannot continue developing this way: the first is the fact that infinity is not an engineering term; the other is that such a solution will be too complex; so normally we stop here and we either develop the control unit as a computer or we develop the control unit as hardwired logic.

Normally, we do not divide the control unit this way but in another from the point of view of the parts we will use, the chips we will use. In this way of thought we have the clock. It is easier to put the clock generator in a chip, a device or system that will generate the clock as we need it. Then, the program controller, that will do everything we need with the program counter. This program controller has the ability to maintain the program counter, increment it, adding something to the program counter, initializing it by loading it from the outside, it will also have something we have not mentioned before, that is some way to save the program counter for the case we want to go temporarily to another location, in what is called a subroutine. The program controller needs then a stack and a stack pointer, to store the current value of the program counter.

The control unit normally includes all the data processing units that the computer will have, as well as their control signals, which make another block. The control unit also has means for controlling memory, the memory address register, the read/write signal, the data input, etc. that, in general, can be considered to form another block. Another block is finally needed to control the operation of all the other blocks and, in this way, we are back into the problem mentioned above of the two parallel mirrors.

Data paths connect the different elements inside and outside the control unit. We have to specify the data paths as elements because every data path needs some form of control; very few data paths are point to point that do not require control; in most cases data paths join the output of several registers to the inputs of several other registers, forming a complete bus. We need control on which unit drives the bus and which one receives the data. So, every data path needs control and that is why we should specify them.

The control unit, in summary, fetches the opcode for the next instruction. It decodes and executes the instruction. It generates signals for the program controller, it generates signals for the data processor, the arithmetic logic unit or whatever other data processor we have; finally, it develops signals to control all the data paths, as well as signals for all the external units.

2.2.- **Control Unit Operation**.

The problem that we are considering now is how to design the control unit. Everything in the computer is sequential. The computer executes one instruction at a time. In order to perform a given instruction we divide the instruction into a series of steps. Normally the clock is divided into phases, one for each step of the instruction to be executed. Let us analyze the steps required to perform some simple operation. The first step is to put the program counter onto the address bus; for this purpose we connect the output of the program counter to the input of the address register and give a pulse: the content of the program counter will be latched onto the memory address register and that address appears in the address bus at the end of the clock; we need a signal that tells the memory that the operation is a read. Then, we need to do something to wait for memory. The previous steps are very short, we have some logic that selects the path and opens the path, there is a delay but it is only a few nanoseconds, 10 to 20 nanoseconds. The delay from the time the address is active until the memory has valid data in the data bus is a much longer delay; typical static memory takes in the order of 200 ns.; dynamic memory takes probably 120 ns. The memory delay is much longer than the delay in the logic. The way we wait for memory is one of the

design decision we must make. The delay to wait for the memory cycle can be developed for example counting five clock periods idle time every time the control unit accesses memory. At the end of this time the data bus is connected to the instruction register and a pulse is given to latch it. Then we increment the program counter probably with a pulse if it is a counter. If the program counter is not implemented as a counter we need to route it through the arithmetic logic unit to increment it. We need some way, depending of our decision, in which we decode the opcode. Decoding the opcode means either logic like in a PLA, or a computer that makes the decoding. In a PLA, the opcode mixed with the phases of the clock produce as output the different signals required through out the computer.

At this moment we analyze the different steps of the instruction. Consider loading a register with the content of memory. We need to put the new value of the program counter into the memory address register, again say read, again count several clock cycles to give time for the memory to develop the data. We want the data to go to the bank of registers, the data goes to the input of the registers, we need to take the address of the register that is normally part of the instruction, and connected to the address input of the register bank, and we should tell the registers to write. We need to increment again the program counter, whatever way it is done, and finally we have to signal the end of the instruction and the fetch of the next one.

So this is the execution of the instruction that in assembler language could be MOVE #N,R1.

As a list, the steps for the FETCH of the next instruction are:

- Connect program counter output to memory address register input;
- Latch memory address register;
- Send memory read signal;
- Wait for memory ready;
- Connect memory data output to instruction register input;
- Latch instruction register;
- Increment program counter.

The steps for the analysis of the instruction and execution phase, considering the instruction MOVE #N,R1, are in list form:

- Connect program counter output to memory address register input;
- Latch memory address register;
- Send memory read signal;
- Wait for memory ready;
- Connect memory data output to register bank data input;
- Connect instruction register output to register bank address input;
- Send register bank write signal;
- Increment program counter;
- Signal end of instruction.

2.3.- **Sequential and Parallel Operation**.

Now we can start looking and how we perform the functions we need to perform. First, it is clear that there is nothing sacred about the order in which the different steps have been ordered. For example, the relative ordering of the read write signal becoming active with respect to the other signals is not that important, it can be changed. Probably more interesting is the fact that it is not necessary that each one of these operations is performed independently of each other. Nothing prevents us to connect the path from the program counter to the memory address register, and at the same time, connect the same signal to the clock input of the memory address register, and at the same time produce the read signal for memory, all in the same step. So, here we have a very simple example of the possibility of abandoning the sequential computer. You remember there are two main characteristics of what we call the von Newmann computer: one is that it has the program stored in memory and the other is that it executes instructions sequentially. All computers today are of this type. Now we are looking at the possibility of a computer that could perform functions in parallel. Many operations can be performed as the same time. It will be wasteful to use one clock period for each one of the operations we have described before. We should try to find all the operations that can be performed in parallel. So, this control unit is showing us one way of improving the computers of today, a mixture of sequential and parallel. It cannot be totally parallel because we cannot do everything at the same time; for example, we cannot perform two accesses to memory at the same time; we have to wait for the end of the first one and then do the second. So, we have to be careful about the selection of the steps, so we can have the maximum possible number of them in parallel. We will see how we end up with a very large parallelism.

Another example of parallelism is putting the address to the register bank; the only condition is that before the data comes the address should be ready. Even from the beginning we can connect the address to the registers, any moment we want. It can even be directly hardwire if we do not need it for anything else.

2.4.- **Hardwired Logic**.

The original way to develop the control unit was to hardwire the logic. The idea is to develop a circuit. Let us look at this method so you have an idea. Imagine we have the instruction with 5 bits in the opcode. These five bits can go to a decoder that gives 32 outputs one for each opcode. We develop a table of steps and determine the maximum number of steps of any instruction. The clock then drives a counter and the outputs of the counter are connected to a decoder that gives a pulse for each one of the steps. Then, we analyze the required signals. When the clock is zero we need a certain path enabled. In this way we AND the clock and the opcode decoder signals and get a single signal. If we have several conditions where we need to enable the same path, we OR all the signals together to get the signal we apply to the path. In this way we develop all the signals that we need for every one of the instructions, for every clock, and then combine them together to form the output signals. You can imagine there are going to be quite a few of these signals. We can put them in a PLA.

We engineers have the habit of testing our work after we finish it. But the problem is that when we check

our work we find ways to improve it and so, imagine we find we can shorten an instruction. How can we do it in this type of design? The problem of this method is that you have to re-design the whole thing. You have to change practically the whole PLA to make a little improvement.

Engineers normally also do something else, we make mistakes. To correct a mistake in this type of design we have to re-design practically all the system, the whole PLA. At the beginning this was the only way to design the control unit. Now we have something else. Before we leave, it is good to ask us if there is any advantage in this type of design for the control unit. The advantage is that this is the fastest possible computer we can make. You have to realize that in this design there is no overhead, no wasted time. Every instruction executes as fast as possible. So, when we need to design the fastest possible control unit we have to use this design.

2.5.- **Microcoded Logic**.

Most of the time we do not want to pay the high price of the development of the control unit; so, what is the other alternative? The other alternative is to put a very simple and very fast computer to organize the operations of the control unit. This computer has an input from the outside world with the instruction register that goes through a ROM that gives the computer the address in its own memory where the execution of the instruction starts. The computer puts out this address and the special memory gives out the program for the computer. The computer executes the instructions one at a time, and every time it executes one instruction, the memory where the instructions are, has another part that is naturally selected at the same time than the program memory. This part produces all the signals we need all over the computer. We have in that memory one bit for every signal we need for the computer. For example, one bit can be the read/write signal to memory; this bit will be one when we need a one and a zero when we need a zero. Another bit can indicate the program counter to increment, and the same with all the other signals. This is what is called the microcode. This is a very bad name because it gives the idea that it is a program. This microcode is not a program, it is harware, as hardware as a PLA. The microcode cannot be considered a PLA although for each bit we develop a function of what we want out at every instant of time. It is more like a program memory and a PLA connected together, sharing inputs, so that what is address for the program memory are input values for the PLA. We talked about two very interesting things, mistakes and improvements. If we have to make a change the only thing we have to do is to change the bits in the ROM the way we need them. The ROM's are not that expensive.

There is something that is important to understand and is that the microcode is not a program. This comes from the fact that the position of the bits does not mean anything. Is the same as with the gates. Imagine that when we design a system we determine to use gate 1, 2, 3 in our circuit. Imagine further that the moment we wire the unit we change our minds and we use gate 4, 5, 6 instead. This change will not change anything in our circuit. The same thing happens with the microcode. Which bit we use for a particular function does not matter as to the operation of the unit. The only thing that matters is what is stored in the ROM for that bit, whichever it is. It is to be noted that the microcode has two parts, one is a program, the program for the sequencer; the other part is the control part, that is not a program. The computer advances the addresses one by one in order, so the computer is normally called the sequencer, or the microcode sequencer.

2.6.- **Synchronous and Asynchronous Operation**.

So, we have two possibilities for designing the control unit. For both of them we have a nasty problem that is probably the less appealing part of the design of a computer. This is the problem of finding out the time delay in the different paths. This permits us to select a clock frequency. We can also think of eliminating the clock completely and each signal is fed back to the counter, producing an increment after the proper delay to permit the propagation of the signals. That will be a very fast computer. This is what is called an asynchronous computer. For example, a timer is started any time the unit accesses memory, after the proper delay a signal is produced and sent back to the counter. Naturally, if we have several types of memory, we will need several timers to produce the delays. So this is a possibility. It is possible to work totally asynchronous.

The advantages of this kind of work is that this is the fastest computer we can develop with a given architecture and physical devices. Note that this computer will work almost as fast as one with wired logic. There is nothing that is overhead, everything has been eliminated. The disadvantage is again, that this is very difficult to change, update, or correct. At the other end of the scale we have a computer where every step of the microcode takes the same length of time, in what is called the synchronous computer. We solve the problem the way it is solved in the microcomputers where the length of time is long enough so that every signal can propagate through its path. If you remember the 6502, all clocks are the same, one microsecond, so you can work with memory with access time up to 450 ns. If the memory takes longer than that, it simply cannot work, if the memory take less than that, the computer must wait idle. This is the other extreme.

There are many different ways to work in between these two extremes. For example, we can have a clock period such that all internal operations are performed in this clock period but the clock period is not long enough for the external operations. The external operations are performed asynchronous by waiting for an acknowledge. This is the way the MC68000 works. The way to implement this is as follows: The sequencer has an input for the increment of the address counter. This signal can be the OR of a bit of the microcode and the acknowledge; when both signals are low the sequencer does not increment the program counter and waits in the same instruction. When the acknowledge comes the signal goes up and the sequencer advances. When it is not necessary to wait, the microcode bit is a one and the sequencer advances continuously. This will be a semi-synchronous computer. This speeds up the operation substantially but this is not the only thing we can do. Imagine that the microcode produces the data and it is latched in the pipeline register; after the data has been developed it has to go through the sequencer to develop the next instruction to be executed. This delay is not that long. On the other hand we have for example, a signal must go through memory, the delay will be much longer. So we have some paths where the delay is short and some paths where the delay is long. A way to work is to go through all the paths in our control unit and calculate the delay for each one of them. If we have a single clock period, it has to be equal to or longer than the longest delay in any path in the unit. Now, when we analyze the different steps, for each one we have to allot a time at least as long as the delay through the longest path in the step, not in the unit. It is possible, that we have several steps that take about the same time, others that group around another time, and so on. In this case, we could have a clock generator that produces

different periods, we can speed up the computer by developing some code from the microcode, some bits that are fed into the clock generator to determine the length of the clock for that particular microcycle.

So we see that from the totally synchronous device to the totally asynchronous device we can have many intermediate possibilities. In general we will use a clock and we will develop some form of semi-synchronous arrangement for our unit to get it to perform as fast as possible.ngest delay. So the only thing we have to do additionally to get some more speed from the computer is to group them and give the commands to the clock generator. Actually, the device we use for clock generator produces variable clock periods so the increase in speed is almost for free. The time analysis is one of the important parts of the design of a computer. Time analysis is a way you can speed up your computer but, more than anything, this is the way you can assure your computer will work. This problem is independent of the way you develop your hardware; depends only on the type of computer you are developing. Naturally, the difference in computations is large between a hardwired or a microcoded computer.

There is a final point to mention and it is that if you have to develop a hardwired unit, you have at this time the necessary knowledge to do it. There is absolutely nothing there that you do not know. This is not true with the sequencer. This is another reason we will concentrate on the sequencer.

The important point about the sequencer is that it is so simple that it is already designed and even in the case you do not use an integrated circuit as sequencer, the hardware of the sequencer is simple and it can be designed and built simply.

2.7.- **The Bit-slice Concept**.

There is one term we need to clarify, it is the term bit-slice. Bit-slice is not an architecture of computers, bit-slice is only a technology. Imagine we have the memory address register with 32 bits. We find that there is no register with 32 bits in the market, but only with 8 bits. We take four of these registers and use them for the memory address register. This is the concept of bit-slice. We do this everywhere in the computer. The arithmetic logic unit is developed as slices of four bits and we can put as many as we want to develop the word length we need. The same with the program controller and with all the difference devices.

The advantage of the concept is that we have a very large family of devices developed for this purpose. These devices have inside ECL logic that is very fast, this logic is surrounded by TTL logic and when we use these devices, we see only the TTL logic. So from the outside is TTL, the driving power, etc. The inside has the speed of the ECL. The interesting part is that there is all kind of devices to build computers. Naturally, the power consumption is quite large, as in any TTL or ECL device.

THE DESIGN OF COMPUTERS USING BIT-SLICE TECHNOLOGY

Michael E. Valdez, Ph.D., P.E.

SLICE #3

Slices: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9

THE MICROPROGRAM SEQUENCER

The material we study in this Slice, is really the most important part of this course. We will be looking into the microsequencer unit and how to use it. We mentioned before, that the most important part of the design of a computer and the one that will take practically all our time, is the design of the control unit. We will concentrate on the design of the control unit using microcode, then we need to study the element that is called the sequencer. We already mentioned that the sequencer is really a computer by another name. There are several sequencers in the market today, we will study one of them, that is typical; we study the Am2910 microprogram sequencer.

3.1. **The Am2910 Sequencer**.

The sequencer is a computer. Please see Fig. 3.1 for a block diagram of the Am2910. Fig. 3.2 shows a normal connection of the Am2910 with all the devices that are normally required, the instruction register, the MAP ROM, the Vector register, the microcode ROM and the pipeline register. These elements are marked on the figure and will be studied later in this slice.

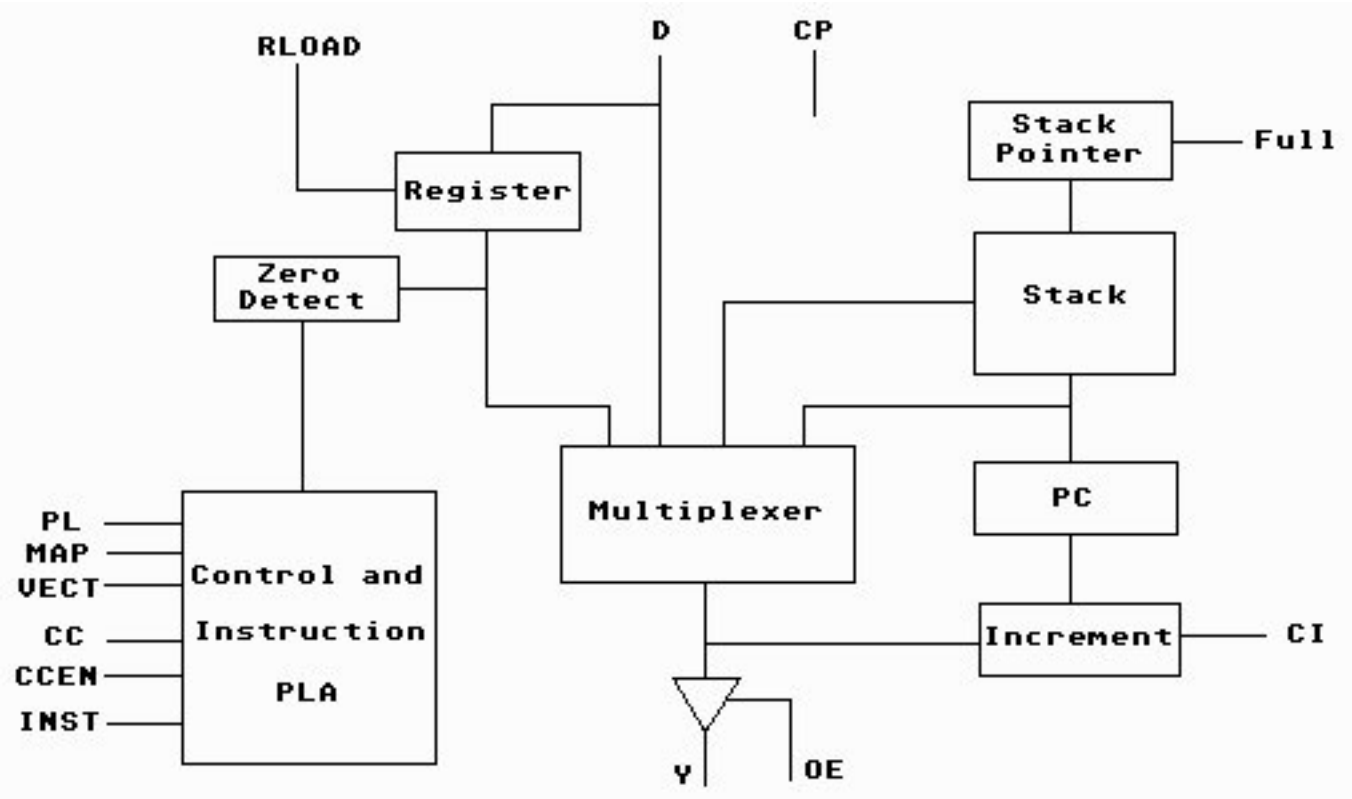


Fig 3-1

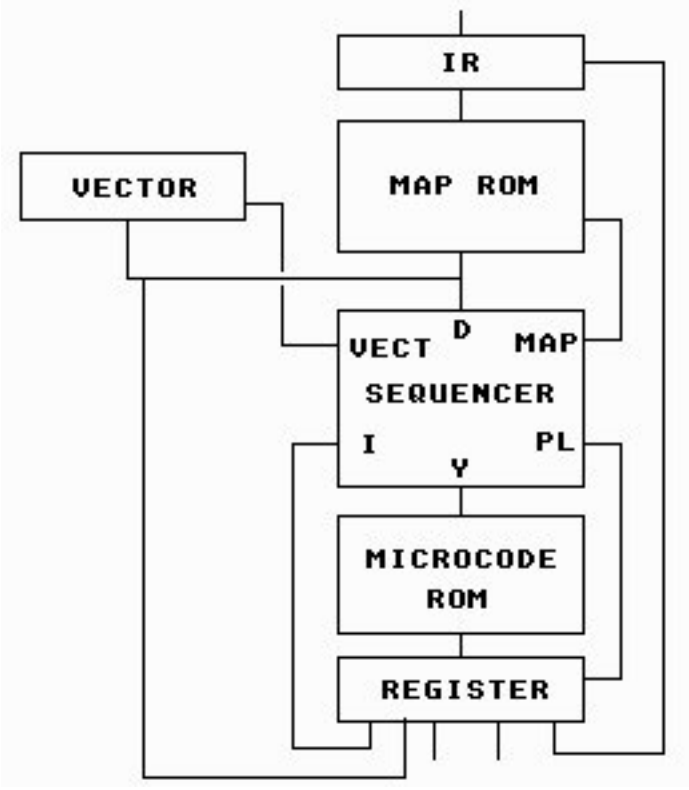


Fig 3-2

This computer has a register, a stack, a stack pointer, a zero detector to find out when the register becomes zero, in the center of the diagram we have a multiplexer that permits us to select the next program counter from four sources: the current program counter, the top of the stack, the content of the

register, or the outside. At the bottom of the block diagram we have the output tri-state devices that are not too useful in this chip because this output is normally connected to the ROM; this control is useful when we have another way to produce the address of the microprogram ROM; for example, if we want to develop a single cycle interrupt. To the left hand side of the diagram, we have the instruction PLA, the input pins to the PLA, the enable of the condition codes, CCEN, and the condition code input, CC.

The program counter that is sent out, goes back through the incrementer, that adds to the program counter the content of pin CI; by putting zero in pin CI we can hold the sequencer executing the same instruction. This is useful for asynchronous devices when we hold the sequencer until a signal comes indicating the external device is ready. There are four sources for the next program counter:

- the previous program counter as stored in the microPC;
- the value sitting on the top of the stack;
- the value stored in the register/counter; and
- the value at the D input pins.

The stack in this device operates in a different way that the memory stack in most computers. The most important characteristic is that the top of the stack is not destroyed when it is used; in this way, the top of the stack can be used for return to the begining of a loop, repeatedly until the loop is finished. This is very convenient for loops but creates the need to destroy explicitly the top of the stack when it is no longer needed.

There is something else in that chip, that is three outputs one bit wide, that are called PL, MAP, and VECT; these bits are intented to enable one out of three devices feeding data into the Am2910 input D. This input is used to give the next program counter when the multiplexer is in D, or to the register, when we give a pulse to register load, RLD. The way this works is by using tri-state devices for the pipeline register (the output of the microprogram ROM), the output of the MAP ROM, and the register that contains or receives the vector for the interrupt. These devices are connected in parallel to the D input of the Am2910. As the microinstructions are executed the Am2910 produces signals to enable one of these three sources to produce the next program counter. This is the significance of the last column of the list of instructions. The signal PL, that enables the pipeline register can be considered as the normal state of the system. Naturally, it is possible to use these signals for other purposes.

Pertinent to the discussion above, we should remember that the microcode has two parts: the microprogram and the microcontrol. The microprogram has two parts: the instruction and the address. The microcontrol has many bits depending on the signals that are needed in the rest of the computer.

The conditions codes are quite interesting. The condition fails when CCEN=0 AND CC=1; the condition passes otherwise. So, we can make the condition pass by setting CCEN=1 or by making CC=0. To assure a fail we need to have CCEN=0 and at the same time, CC=1. So, it is easier to produce a pass than to produce a fail.

The input that is applied to the pin CC in general comes from a multiplexer whose select lines are connected to the pipeline register or to the instruction register. The CCEN could be from the pipeline register to force a pass or fail as required. It is clear that the multiplexer for the selection of the condition could have as many inputs as required. The condition codes comming from the arithmetic logic unit normally produce directly eight conditions when we consider only the set or reset of all the values; i.e., carry set, carry clear, zero, non-zero, etc. If we want to have more complex conditions, and when we want to have the possibility of forcing a zero or a one into the pin CC, then we need many more possibilities in the multiplexer. We also need the logic for developing the conditional equation we want to use. This is the case when we need relations like larger than, etc.

Something that is always very good and nobody ever gets too much practice on it, is reading the specifications. There are three chips in the specifications: the Am2910, the Am2910-1, and the new Am2910A. Either one supports 100 ns per microinstruction. This means that it is possible to perform 10 instructions per microsecond; this is pretty fast! It is very difficult to achieve this; normally you get from 130 to 150 ns per instruction. What we are saying is that in a fast system, the chip can go at the rate of 100 ns per microinstruction.

The DC characteristics show that the device works with TTL levels. The big difference between the Am2910 and the Am2910A is that the later is built with the new IMOX technology that puts ECL inside the chip and TTL in the peripheral. On the other hand, the Am2910 and Am2910-1 are all TTL. Something interesting that you should consider is the last line, Icc, the power supply current, that is around 300 mA per chip. This can be compared with the NMOS microprocessors that take in the order of .5 mA. Of course, this devices can drive 10 TTL loads, while NMOS devices can only drive one.

3.2.- **The Instruction Set**.

The instruction set of the Am2910 is summarized in the following table. Note that the first column marked Reg refers to the conditions of the register that affect the outcome of the instruction; the second column marked Reg refers at the action taken with the content of the register. The condition fails when CCEN is low AND CC is high; the condition passes when CCEN is high OR CC is low. When the condition fails or passes, the sequencer puts in the outputs Y the value indicated under its column and the stack takes the action indicated under its column. The instructions are:

Instr	Name	Description	Reg	Y Stack Fail	Y Stack Pass	Reg	Enabl
0	JMPZ	Jump Zero	x	0 clear	0 clear	hold	PL
1	CJSB	Cond JSB PL	x	PC hold	D push	hold	PL
2	JMAP	Jump MAP	x	D hold	D hold	hold	MAP
3	CJPL	Cond Jump PL	x	PC hold	D hold	hold	PL
4	PUSH	Push/Cond Ld R	x	PC push	PC push	Note	PL

5	CJSR	Cond JSB R/PL	x	R push	D push	hold	PL
6	VECT	Cond Jump Vect	x	PC hold	D hold	hold	VECT
7	JRPL	Cond Jump R/PL	x	R hold	D hold	hold	PL
8	RLOP	Repeat Loop	<>0	F hold	F hold	dec	PL
.	.	R<>0	0	PC pop	PC pop	hold	PL
9	RLPL	Repeat PL	<>0	D hold	D hold	dec	PL
.	.	R<>0	0	PC hold	PC hold	hold	PL
10	CRET	Cond Return	x	PC hold	F pop	hold	PL
11	CPOP	Cond PL & Pop	x	PC hold	D pop	hold	PL
12	LOAD	Load Counter	x	PC hold	PC hold	load	PL
13	TEOL	Test End Loop	x	F hold	PC pop	hold	PL
14	CONT	Continue	x	PC hold	PC hold	hold	PL
15	TWBR	Three-way	<>0	F hold	PC pop	dec	PL
.	.	Branch	0	D pop	PC pop	hold	PL

Note.- During the execution of instruction 4, PUSH, if the condition passes the register is held; if the condition fails the register is loaded from the D-inputs.

Let us analyze the instruction set. We have 16 instructions and we need 4 bits. Most instructions are conditional. JMPZ, which jumps to zero (the next instruction is taken from microcode memory location 0), and clears the stack, is not. This instruction is normally used for the end of the macroinstruction; the fetch procedure is put in location zero of the microcode and we end all macroinstructions by giving the microinstruction JMPZ. The advantage is that this is the only instruction that permits to jump without giving an address; all the other jump instructions require the address of where to jump. An address is twelve bits of microcode; we need these twelve bits the whole depth of the microcode ROM. twelve bits of microcode ROM is something to consider. Now, the instruction JMPZ has a side effect that should be considered and that is that the stack is cleared; that is, nothing can be passed through the stack. Since this is not too often done, there is not too much to worry. On the other hand, the register is held when this instruction is executed.

Most of the instructions are conditional, the instruction 0, JMPZ, is not conditional, or better say, it has the same outcome whatever the condition. The instruction JMAP, number 2, has also the same outcome independently of the condition; this is the instruction that is used to load the address of the macroinstruction that should be executed, after the opcode has been mapped by the MAP ROM into the

address of the microcode ROM where the execution should start. So, the normal procedure is to produce the fetch of the instruction and to give the instruction JMAP. This instruction is very interesting because in many instructions we can find one step where the memory is not being used and we can fetch the next instruction; thus, when the computer finishes execution of the current instruction, the next one is ready in the instruction register, saving the time of the fetch.

Going back to the instructions that are not conditional, we see that we have four where the condition does not change the operation but it changes if the register is loaded or not. In any case, the program counter that contains the address of the next microinstruction is pushed into the stack; if the condition passes, the register is loaded from the D input which is normally fed from the MAP ROM or the pipeline register.

The final instruction that is not conditional is 14, CONT, that is an instruction to do nothing and execute the next microinstruction. This instruction does not change the stack or the register. The purpose of this instruction is to produce control signals we need for a microcycle. This is the most commonly used instruction when developing microcode for computers.

There is another level of conditional instructions that depend on the value of the register. For example 8, RLOP, repeat loop as long as the register is not equal to zero. So you load the register with the number of times you want to do the loop and push into the stack the return address; like with instruction 4, PUSH. Using instruction 8, RLOP, if the value of the register is not zero, it decrements it and goes to the location indicated by the top of the stack. If the register is zero the sequencer executes the next instruction, and pops the stack, destroying the previous value. Instruction 9, RLPL, is similar but the next instruction at the end of the loop is taken from the input D rather than the stack. These two instructions provide us with the means to produce loops for executing a given process a fixed number of times.

Note that the instruction that reads the stack does not change the stack pointer. The top of the stack is always available to the multiplexer in what is called F. This is a different way of operation of the stack than in a microprocessor, where a pop instruction takes the top of the stack and puts it some place; in a microprocessor there is not way to copy the top of the stack without modifying the stack pointer. Note that the way it is done in the Am2910 is very efficient, since we do not need to make any operation to read the top of the stack.

The three-way branch is a instruction where we have three possibilities. If the register is not zero, it decrements the register; what happens after that depends on the condition code where it will go to the stack, F, if the condition fails and to the program counter if it passes. If the register is zero, the choises are different: it goes to the value presented in D if the condition fails or to the program counter if it passes. Something else to consider is that the stack is poped in all cases except when the register was not zero and the condition fails. So, this instruction provides us with the possibility of executing a loop for not more than N times, or until a certain condition is met.

Now, with respect to the other instructions, we have the normal jump to subroutine with many possibilities. All of them are conditional. Instruction 1, conditional jump to subroutine to PL, gets the

address from the pipeline register and the jump happens if the condition test passes and the next instruction is executed if the condition fails. Instruction 5, CJSL, is a conditional jump to subroutine that is always executed; that it, the jump is taken any way, and the condition determines if the address comes from the pipeline register or from the register.

We have the conditional return meaning that the computer takes the next address from the stack and pops it if the condition passes or from the program counter and leaves the stack alone. It does not return from the subroutine if the condition fails.

We have several jumps, with the instruction 0, JMPZ, jump to location zero that was studied before, the instruction 2, JMAP, that was also studied before, the instruction 3, CJPL, that is a conditional jump PL, meaning that if the condition passes the next address comes from D, if not it is the current program counter. Instruction 6, VECT, is a conditional jump to vector, which jumps to the content of D and the signal VECT is active only if the condition passes; if the condition does not pass, then the instrucion is like a continue or no operation. This is the basic instruction to serve interrupts. The interrupt should make the condition pass and it is served by this instruction. The typical way to use the interrupt between macroinstructions is to test the interrupt as part of the fetch of the next instruction. Then, the fetch instruction is started with the instruction 6, VECT, conditional jump vector; so, if there is an interrupt, the interrupt is served; if there is no interrupt the fetch proceeds as normal. This is the interrupt at the macro level; later we will see how an interrupt could be produced at the microintruction level. Naturally, we pay for this capability by adding one micro instruction to every instruction. We have a jump R or PL, where the computer jumps one way or another, depending on the condition and finally, a conditional jump PL and pop, that, if the condition passes, the unit takes the next instruction from D and at the same time, pops the stack; that is, destroys the top of the stack and gets up the next value. This last instruction permits us to develop loops that run until an external condition is met.

We have other instructions that are neither end of loops, jumps, or jumps to subroutines. Instruction 4, PUSH, pushes the stack and depending on the condition code it loads the counter or not. Instruction 12, LOAD, only loads the counter unconditionally and continues. Instruction 13, TEOL, tests the end of the loop, for the loops that do not depend on the counter; with this instruction the Am2910 tests the condition code, if the condition passes it ends the loop by poping (destroying) the stack and continuing execution; if the condition fails the Am2910 takes the address for the next instruction from the top of the stack. Finally, we saw that instruction 14, CONT, is a do nothing instruction that is useful when the important part of the microcode is the control part instead of the microprogram.

3.3.- **Pipelining**.

This is a very important concept that we need to analyze. Imagine we have a hardware loop formed by combinatorial logic. The only restriction is that there is no storage device. Imagine that some how, turning the circuit ON for example, we put a pulse in this loop. Even in the case of lose of energy, the gates restore energy into the circuit and the pulse propagates through the loop forever. How long it takes to perform a cycle in the loop will depend on the total delay on all the combinatorial logic.

This is not too useful but introduces us to the subject. Imagine we open the loop and put a register. This register has a clock input and when we give a pulse to the clock, it latches the data from the inputs to the outputs. There is a small delay from the clock to the outputs stable. Since the external logic is combinatorial, the output data starts propagating immediatelly through the loop until finally gets back to the input of the register. The basic question is how often we can give the clock? It is clear that we cannot give the second clock until we have given enough time for the signal to propagate through the whole loop from the outputs to the inputs of the flip-flop. If that is the clock of the computer, this clock determines how fast the computer can work.

The fundamental way to speed up the computer is what is called pipeline. What we do is to break the loop we have by installing another register, ideally at the point where the propagation time is half of the total. If we have a second register, the signal that is latched in the first register has to propagate only to the inputs of the second register; the data that is latched in the second register will propagate at the same time to the input of the first register. What we are doing is speeding up the computer by performing two operations at the same time. These two operations could be two steps of the same instruction or two separate instructions. In this way the clock can go almost at twice the previous frequency. It is important to note that now it takes longer to go around the loop, because of the delays introduced by the registers, but this does not have any effect on the operation of the unit.

This process could be continued as far as practical. If there are other positions where it is practical to break the loop, we can pipeline more and in that way, we can have more parts of an instruction executing simultaneously, in parallel. It is interesting to note that this is not the way the normal macrocomputer works; we have mention before the normal computer is a sequential machine that performs one step at a time.

There is a limit to this pipelining process and the limit is that it must be possible to partition the instruction into steps that can be performed at the same time. There are very natural situations where we can break the loop in parts and is when the loops are simple; there are others situations where the resources mix into different loops and then we must be careful before cutting the loops for pipelining.

There is another case where pipelining has trouble, and it is the problem of the conditions; that is, the function the next instruction will perform depends on the outcome of the previous instruction, and that outcome is not known until very late in the execution of the instruction. In this case, it might be necessary to destroy the data in the pipeline and start over from the beginning.

On the other hand, if a sequence of instruction is independent of each other, we could pipeline them very much to the extent that the net time for the execution of each instruction becomes only one period of the clock driving the microsequencer.

This shows us that the advantages of pipelining have a limit, as normal. Register to register instructions can be overlapped very much, when the next instruction is the proper instruction. Imagine a sequence of ADD register to register instruction, they can be packed very much as long as the registers used are

different or in the right order. But now imagine that we add the content of several register to the same register; in this case it is not possible to perform the second ADD until the first one has ended.

Imagine also that the second instruction is a branch on condition, as in the case of a multiple precision ADD, then this instruction cannot be executed until the end of the ADD. The problem is that the computer has to go and test the next instruction in relation with the current instruction, to see if it is possible to pipeline it or not. This test takes time and, in general, slows down the normal operations of the instructions. If what we gain by pipelining is less that what we have to pay for the tests, then it is not logical to do it.

What is easy and in many cases there is no problem, is overlapping the steps of the instructions. The difference is very important. When we design a computer we define and set forever the steps of the microinstructions. On the other hand, the user of the unit has the freedom of inserting instruction in any order he wants, that in general we cannot foresee. Then, the overlapping of microsteps requires only that we see if the next microstep can be overlapped or not.

One very typical example is the execution of the fetch for the next instruction. We can go, instruction by instruction, and find a moment when the address and data buses are not used to access memory and load the next instruction into the instruction register. In this way, the moment the instruction under execution ends, we can start with the next. Since the fetch takes at least two microinstructions or steps, the savings can be substantial, in the long run.

In the circuit of the sequencer there are typical places for establishing pipelining and are at the output of the microcode ROM, so the next address can be propagating while we execute the current step; the instruction register mentioned above; and sometimes the output of the MAP ROM.

In the memory circuit there are several places where we can establish pipelining and is in the address lines using what is called the memory address register, in the data lines, whether for storage into the memory or for reading memory. These last registers require analysis of the special conditions of the design.

The other side of the coin is when we are forced for some reason to use a device that is slow; for example, using a slow ROM for the microcode. In this case, it is easy to realize that the slow ROM will control the frequency of the clock and then it does not pay to spend a lot of time trying to develop pipeline in other parts of the circuit since, at the end, the circuit will be waiting for the clock. Remember that pipelining is not done for free; there is some analysis to make before developing pipelining and this analysis should be made where there are reasonably expectations that we will succeed.

A final note about pipelining is to recall your attention to the fact that there are two possible ways to implement the control unit of a computer: one is hardwired and the other is microcoded. There is a big difference between the two in the possibilities for pipelining. In the hardwired version it is very difficult to develop meaningful pipelining inside the control unit. Pipelining can only be develop outside, in the

memory circuit for example, but in the control unit is difficult. This creates an idea that although the hardwired unit is intrinsically faster than the microcoded unit, by using all the possibilities of pipelining in the microcoded unit, we could make it work faster than the hardwired one. The advantages of design and maintenance of the microcoded unit will then be a dominant factor for chosing this design.

Normally, it is considered that there are three standard places for pipelining, the instruction register, the memory address register, and the memory data register, and there are classifications of the architectures with respect to this point. From the point of view of the internal loop of the control unit, these three places are with respect to the microcode ROM. These two loops are not independent of each other, so the definitions get blured and are not that useful.

3.4.- **Time Analysis**.

When we design a control unit, a moment comes when we have to define the clock period. At this moment we perform two very important functions, one is the definition of the period of the clock, whether synchronous or asynchronous, and at the same time, we perform a very complete checking of our design. The reason is that in order to compute the period of the clock we have to look through all the possible data paths in the circuit and compute the time delay in each one of them. At this moment, you have enough experience in designing (probably more than you realize), and given a set of specifications you can put blocks together to perform the functions that are required. Once you get that on paper, the problem comes of how to know if the unit works and how it works. When designing microcontrolled units we have a basic tool that is the time analysis, or the delay analysis.

What we do is to go through our design and mark all the possible paths for the propagation of the data; especially those that belong to the pipelines. We should look at all the places where data or control signals are latched, to analyze where they have to go before we can use them. The analysis of these paths will provide us with the period of the clock, among other things. The mechanics of doing it is to get many copies of the diagram (the block diagram is enough) and mark on them the different paths, one on each copy.

Let us look at the simple block diagram of the sequencer, as shown on Fig. 3.3, and see what loops we have. We have the instruction for the sequencer comming from the pipeline register, to the Am2910, passes through it to the Y output, then through the microcode ROM, and the pipeline register where it latches. This is then a closed loop.

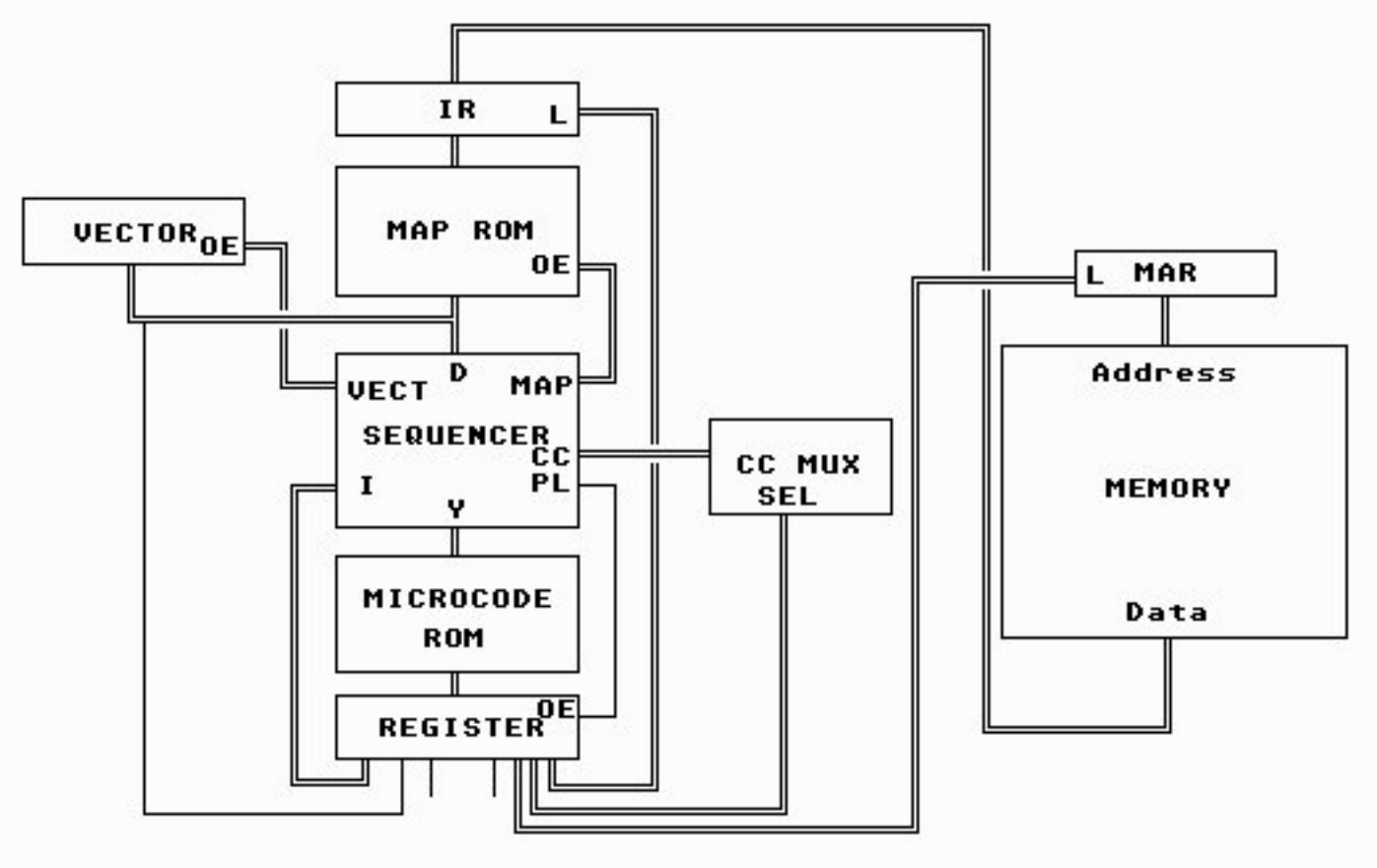


Fig 3-3

Another loop is from the pipeline register through the condition code multiplexer, through the Am2910, through the Y output, to the microcode ROM and ending in the pipeline register. None of these loops have any pipelining. They affect directly the clock.

Another loop is for the address of the next instruction. The loop starts in the pipeline register, goes to the Am2910 through the D input to the Y output, to the microcode ROM and the pipeline register, again without any pipelining.

We have to analyze paths like the vector in which we have to assume that the condition code for the interrupt passes and the Am2910 goes to read the vector. After recognizing the instruction and testing the condition code, the Am2910 produces the signal VECT; the vector address comes from the register to the D input, to the Y output of the Am2910, to the microcode ROM and to the pipeline register.

The instruction MAP is similar. The Am2910 receives the instruction from the pipeline register, produces the signal MAP, the content of the Instruction register is driving the MAP ROM so we have to consider only the delay in the ROM from the moment the output is enabled until the value goes to the D input of the Am2910, to the Y output, to the microcode ROM and ending at the inputs of the pipeline register.

There is another loop that is of interest, this is the loop that involves memory. Consider the following path: from the pipeline register a signal goes to the memory address register, ordering it to latch; the value at its inputs is stored there and propagates into the memory devices as an address for the

instruction, the data output from the memory propagates to the input of the instruction register, where it is latched. The content of this register propagates as an address into the Map ROM and then, with the instruction MAP, the data output from this ROM propagates into the 2910, entering through the D-inputs and going out through the Y-outputs; this output is used by the microcode ROM as an address and the data outputs from the microcode ROM go to the input of the pipeline register, where it is latched, closing the loop.

In order to make this analysis we use the data the manufacturer gives us as switching characteristics. The Am2910 has tables where the composite delay from input to output has been combined into a single value. This is done for commercial and military units, for each one of the devices.

The same can be obtained for all the different devices we use in our system, using the data sheets of the manufacturers of the devices. By adding the time delay around the loops we get the total delay on each path. The value we obtain gives us a measure of the quality of our design, the advisability of trying improvements, etc. We will get into details later.

THE DESIGN OF COMPUTERS USING BIT-SLICE TECHNOLOGY

Michael E. Valdez, Ph.D., P.E.

Slice # 4

Slices: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9

USING THE SEQUENCER

This slice is devoted to the analysis of the use of the sequencer to perform functions, how the operations are analyzed and the micro code produced, how the additional hardware is identified. We will mainly be concerned with the use of the Am2910 but some alternatives will be studied in the final paragraphs of the slice. More of these alternatives will be studied while solving later examples that refer to a computer and when we analyze other devices that are available in the market.

4.1- **Straight Code and Overlapping**.

The simplest case to study is when the code we need is only straight, without loops, subroutines or any possibility of overlapping. We will study the case of the typical fetch of an instruction in a computer. We saw in another slice that the fetch requires the following steps:

- Transfer the content of the program counter to the Memory Address Register;
- Signal memory read;
- Wait for memory;
- Transfer the data out from memory into the Instruction Register;
- Increment the program counter.

At this moment we assume we have the hardware described in previous slice, figure 3.2. That circuit

does not show any program counter, memory address register, or any memory; it does show the instruction register. With the addition of these elements, the circuit becomes Fig 4.1, where we have already added data paths required for the transfers mentioned above. The circuit also shows as black dots the points where special signals are required, signals that is our responsibility to generate. Let us face this problem.

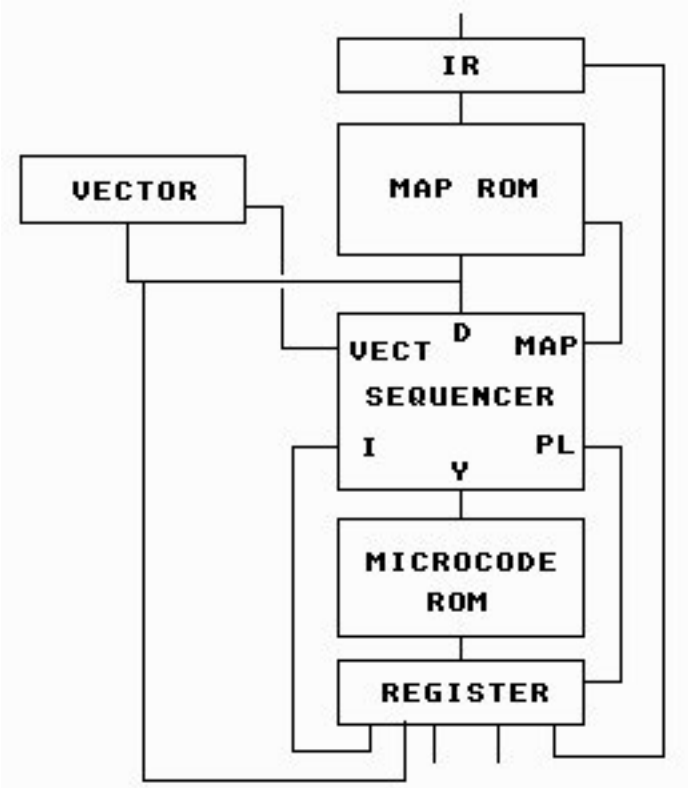


Fig 3-2

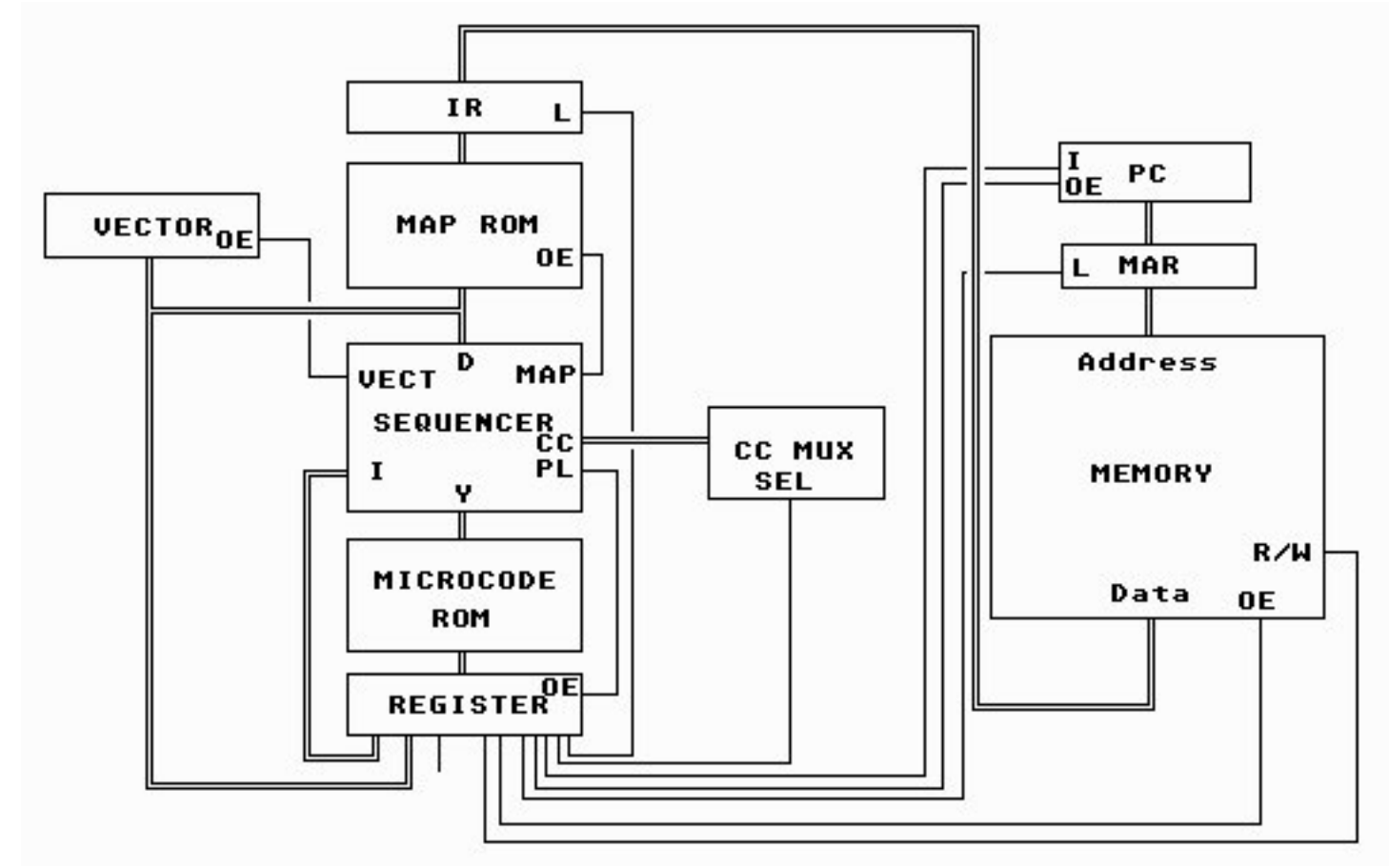


Fig 4-1

The transfer of the content of the program counter to the memory address register requires to enable the output of the program counter and to signal the memory address register to latch the data on the bus. In the simple circuit of Fig 4.1 we do not need to do anything with the bus, since there is a single bus. It is possible in other cases, that it will be necessary to enable a path through a complex bus. The signal to indicate to memory that the operation is a read has already been mentioned. Since the speed of memory is much slower than the speed of the registers, we need some way to wait for memory to be ready. Finally we need to enable the output of memory into the bus and signal the instruction register to latch what is present at its inputs. One point that is important is that when we mention signals, we assume that all the signals we do not mention as active, are inactive. For example, the signal that enables the output of the program counter is active only while it is needed; in the next step we do not say anything about it, so it is inactive. This is a general rule we will use in all our descriptions.

So we see that although our problem is simple, when we analyze it, it becomes more complex; the steps really needed for the fetch of an instruction are then, considering what we have just said:

- Enable the output of the program counter;
- Signal the memory address register to latch;
- Signal memory for read;
- Wait for memory;
- Enable memory output;

- Signal instruction register to latch;
- Signal the program counter to increment.

Let us look at how we can generate these signals to produce the effect we want. Imagine that, following normal practice, we put the fetch at location zero of the micro code read only memory. We should recall that the micro code read only memory has two parts, the micro code program and the signals. In the part of the signals we need bits for the enable of the output of the program counter, to latch the memory address register, to signal the memory it is a read, to enable the output of memory, to latch in the instruction register, to increment the program counter. So we make a table with a heading as follows:

#	Am2910		Signals					
.	Inst	Adr	Program Counter		Mem Add Reg	Memory		Ins Reg
.	.	.	Output	Increm	Latch	R/w	Output	Latch
.	.	.	.	.	.	.	.	.

We will be filling this table as we develop the micro code. We will also add more columns as we find the need for more signals. At this moment, filling this table is straight forward. We will assume for the moment, that a 1 makes the signal active and a blank makes it inactive. Let us start with the signals and we end up with:

#	Am2910		Signals					
.	Inst	Adr	Program Counter		Mem Add Reg	Memory		Ins Reg
.	.	.	Output	Increm	Latch	R/w	Output	Latch
0	.	.	1	.	.	.	.	.
1	.	.	1	.	1	.	.	.
2	.	.	.	.	.	R	.	.
3	.	.	.	.	.	R	.	.
4	.	.	.	.	.	R	1	.
5	.	.	.	.	.	R	1	1
6	.	.	.	1	.	.	.	.

It should be obvious by now, that this list of instructions has many superfluous steps that are really not needed. The idea is to emphasize this point. For example, step #0 is not needed, at the same time the output is enabled we tell the memory address register to latch. The reason is simple, the output of the

program counter is enabled for the full cycle because it is activated by a level; then there is time for the data to come out to the pins, propagate through the wires to the input of the memory address register and into the memory address register ready to be latched. The latch operation on the other hand is activated by a transition; that is, we use a register that is edge triggered and that will latch in the negative transition of the signal, or at the end of the cycle. This will be true for all the latch operation that we will consider.

Steps 2, 3, and 4 are required to give time to the memory the develop the data but step 5 is not really required, data could be latched at the end of step 4 with an edge triggered register in the instruction register, as explained above.

Step 6 requires a separate paragraph because it is a totally different problem. Here we have the need to increment the program counter and we did it at the end of our instruction. The program counter is not needed after step 1, consequently, we could increment it at any moment we want. When we increment the program counter, we do not need any of the resources that are being used for the other operations. So, the logical thing to do, is to use the time while the memory is developing its data to increment the program counter. Notice that this operation is also edge triggered, and consequently can be put with step 1. More on this later.

After we make the modifications indicated above and renumbering the steps, our micro code ends up being:

#	Am2910		Signals					
.	Inst	Adr	Program Counter		Mem Add Reg	Memory		Ins Reg
.	.	.	Output	Increm	Latch	R/w	Output	Latch
0	CONT	.	1	1	1	.	.	.
1	CONT	.	.	.	.	R	.	.
2	CONT	.	.	.	.	R	.	.
3	CONT	.	.	.	.	R	1	1
4	MAP	.	.	.	.	.	.	.

So, this simple example has permitted us to see how we develop the micro code, practically without any effort, how we optimize the code by paralleling operation and by overlapping independent operations.

One final point to indicate, and that produces a lot of problems, is that during part of step 1 the program counter is incrementing, and it cannot be used for a while; for the same reasons, the memory address register latches at the end of step 0 and during the beginning of step 1 the output of the register changes from the old value to the new one. Since this new value is the address of memory, memory does not

receive the new address until the outputs of the memory address register have settled to the new value. We will see this point in more detail when we talk about timing but you have to keep this idea in mind.ry and uses it as an address for the micro code read only memory. These instructions have been added to the table above. The reason for the empty line in step 4 is similar to what was said before about the memory address register, the instruction register only latches the data from memory at the end of step 3 because it is an edge triggered device, consequently, if we put MAP in step 3, the Am2910 will be using the output of the MAP read only memory produced by the old content of the instruction register; that is, the unit will be executing the previous instruction. We will talk about this later. Notice that this micro code assumes we have a MAP read only memory that is very fast, much faster than the period of the clock. If this is not the case, we will need a wait step, as for memory.

Something else to note is the location of the instructions for the Am2910. The instruction MAP for example in step 3 will not indicate that the operations indicated by that line are performed with the new address. What we put in the column of the instruction for the Am2910 affects the NEXT instruction that will be performed. Notice that when the signal to latch is sent to the output register of the micro code, the instruction MAP goes to the instruction input of the Am2910; this instruction is being executed and, at the end, the Am2910 will send to the micro code read only memory the address received from the MAP read only memory. The next step in the micro code will not be then from the next sequential address but from the new address.

4.2.- **Subroutines and Nesting**.

The students are familiar with the basic idea of the subroutines. The Am2910 also permits to perform subroutines at the micro code level. We have only one instruction for the jump to subroutine.

Instruction # 5, CJSB, is a conditional jump to subroutine; what is selected by the condition is the source of the address in the micro code read only memory where the sequencer will jump, if the condition fails next address comes from the internal register R; if the condition passes the next address comes from the input D, which is normally connected to the output of the micro code read only memory. One variation of this instruction could be to use the output of the MAP read only memory if the condition passes, with the system executing the next instruction if the condition passes, or the subroutine pointed to by R if it fails. Naturally, pass and fail are relative terms.

The return from subroutine is also performed by only one instruction which is also conditional. In this case the condition determines if the return is taken or not; if the condition passes the return is taken and if the condition fails the program continues in the next sequential address. This is instruction # 10. Return, in this case, means that next address is taken from the top of the stack and this value is destroyed by changing the stack pointer.

One common problem with subroutines, and its basic advantage, is nesting. It is possible to call a subroutine from inside another. The Am2910 permits nesting to a level of 17, which is quite deep for normal operations. If you consider that the stack is also used for loops, this depth means 17 nested calls

to subroutines and loops in total. The real truth of the matter is that subroutines are not used too much in micro programming and seldom the level of nesting will be a problem. Nesting of loops creates another problem that will be discussed later.

One of the considerations in the use of subroutines is that most of the time the address where to jump comes from the micro code read only memory. This requires 12 bits of micro code read only memory, for the whole depth of the read only memory. In a complex operation, that is when subroutines become valuable, this represents a large increase in the size of the micro code read only memory.

4.3.- **The Use of Loops**.

The loop is probably the most powerful construct for micro code. We call a loop when certain code is repeated until a condition is met. The Am2910 has a powerful set of instructions for performing loops. There are several types of loops; the most important are counting loops and conditional loops. That is, the loop is performed a fixed number of times, like when you put seven buckets of water into a tank. The other type of loop, the conditional loop, is until a certain condition is met; for example, when you put water into a tank until it is full. There is a third type of loop where the operations are performed a certain number of times unless a certain condition is met; for example, you put seven buckets of water into a tank, unless the tank gets full.

The basic instruction to start counting loops is instruction # 4, PUSH, push and conditional load R. During the execution of this instruction the value of the current program counter is pushed into the stack; that is, becomes the top of the stack. If the condition passes the value in the auxiliary register is held; if the condition fails the auxiliary register is loaded from the D input to the Am2910. As we mention before, this input normally comes from the output of the micro code read only memory and thus, it is part of the micro code.

We have another instruction to load the auxiliary register, instruction # 12, LOAD, that loads the register unconditionally. Note that this instruction does not pushes the program counter into the stack.

The Am2910 has several instructions for the end of the loop. Instruction # 8, RLOP, is the basic instruction for the counting loop, it is repeat if R is not zero. The Am2910 tests the register with the included logic and if the value in the register is not zero, it decrements the register and takes the next address from the top of the stack. As we said before, this operation does not destroy the top of the stack. If the register has zero, it pops the stack destroying its top value, and takes the next address from the program counter. The conditional input does not affect the outcome of this instruction. This is then useful when the loop must be completed a fixed number of times.

Instruction # 9, RLPC, works very similarly. The only difference is that if the register is not zero, it take the next address from the D input, i.e., the output of the micro code read only memory instead of the stack. Recall the instruction that permits you to initiate a loop without pushing the current address in the stack.

Instruction # 13, TEOL, test end of loop, is included to permit us to develop loops that end with a condition. If the condition fails the loop is repeated getting the next address from the top of the stack; if the condition passes, the stack is popped and the next instruction is taken from the program counter.

Finally, instruction # 15, TWBR, is provided to permit us the perform loops of the unless type. It provides us with a three ways branch. If the register is not zero, it is decremented, if the condition fails the loop is executed from the top of the stack; if the condition passes the loop is stopped, the stack popped, and the next instruction is taken from the program counter. If the register contains zero, the stack is popped any way and, if the condition fails the next instruction is taken from the D input, if the condition passes, the next address is taken from the program counter.

4.4.- **Nesting Conditional Loops**.

The problem of nesting conditional loops is not as easy as the nesting of subroutines. The general criteria for nesting loops, as with subroutines, is that the inner loop must finish first before the outer one can proceed. There are few problems in the case of conditional loops. Imagine for example, that we are receiving data from a keyboard at the micro code level. The outer loop receives a value and some hardware logic tests if the received value is a carriage return (hexa D), for example. This condition is introduced into the condition code. During the execution of this loop, we need another conditional loop, that will wait for the keyboard active. In this loop, the test for the keyboard will be repeated until the active signal from the keyboard makes the condition pass. During the execution of this internal loop, the status signal from the keyboard is connected to the condition code. Let us analyze in more detail this problem.

The hardware we need involves the basic circuit for the sequencer plus the circuit to connect the keyboard, the circuit to detect the carriage return and the multiplexer for the condition code. We also need memory and a memory address register that can be incremented and loaded. All these elements and their connections are shown on Fig 4.2.

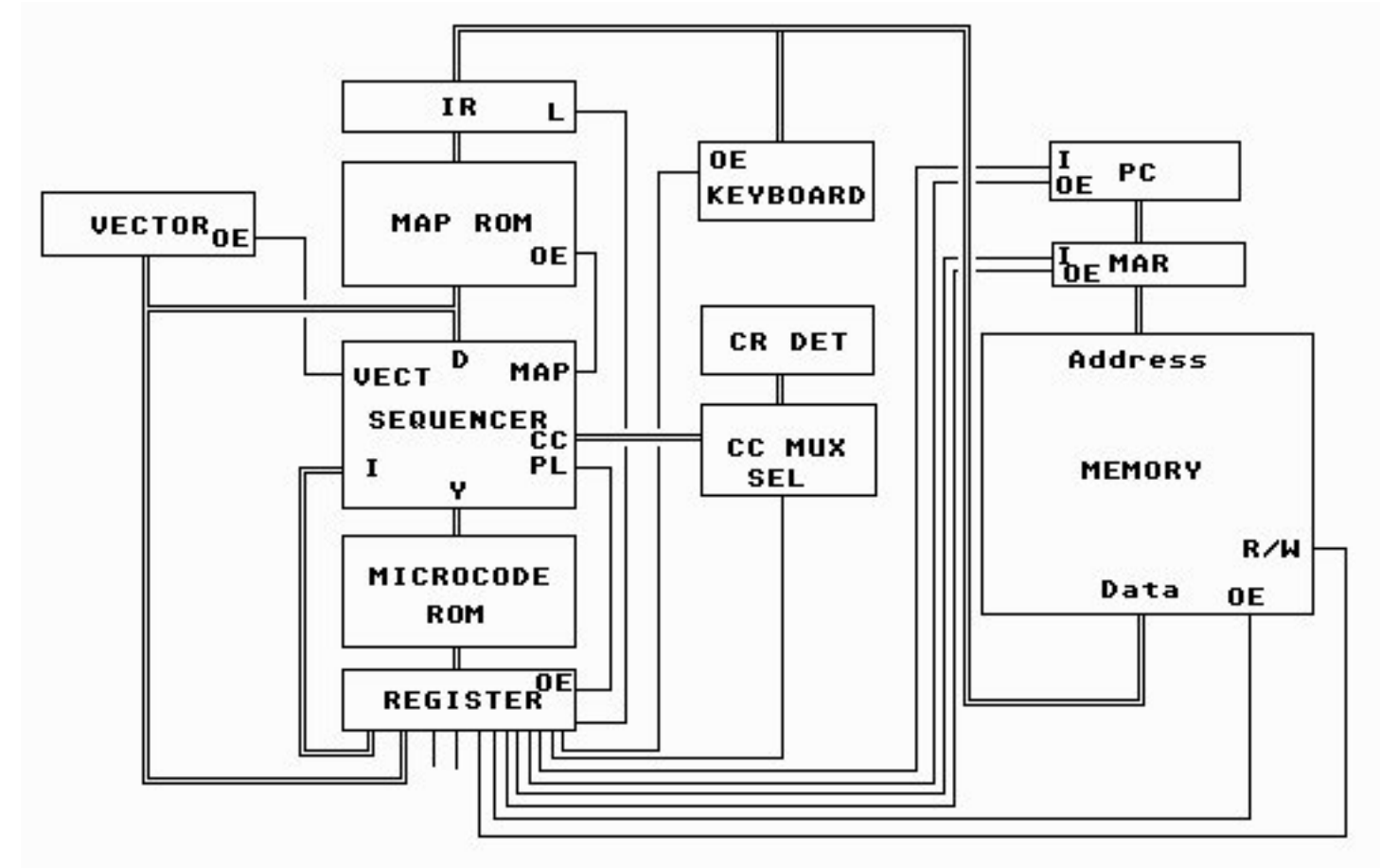


Fig 4-2

The micro code we need for this part of the work will come out as we analyze the functions that will be performed. We start by latching in the memory address register the initial address where we want to store the data coming from the keyboard. The second instruction then pushes the program counter into the stack with the instruction 4, PUSH. Note that although we do not need to load the register, we do not have an instruction to push the program counter without storing the register. If we want to be sure that the register is not destroyed, we need to control the condition code enable bit to force the instruction into not loading the register. We could use one of the jumps to subroutine but we will need to fake the address, which is even worst.

Our next operation is to perform the loop that waits for the keyboard. This, again, requires to push the program counter into the stack with instruction 4, PUSH. The next instruction is the end of this loop with instruction 13, TEOL. Note that during the execution of this instruction we need to move the condition code multiplexer to the side of the status from the keyboard.

When we end the inner loop, we read the keyboard that we do by enabling the keyboard output. Note the active signals to store the data and to increment the memory address register at the end of the cycle. Then we are ready to test the end of loop with instruction 13, TEOL, but our code requires that we switch the multiplexer to the side of the logic that detects a carriage return. The micro code is then as follows:

#	Am2910		Keybrd	CC	Memory Add		Memory
.	Inst	Adr	Out	Mux	Load	Inc	Write
11	CONT	.	.	.	1	.	.
12	Push	.	.	P	.	.	.
13	Push	.	.	P	.	.	.
14	EOL	.	.	K	.	.	.
15	CONT	.	1	.	.	1	1
16	EOL	.	.	C	.	.	.

There is another way in which the same functions can be performed, that is using a subroutine for the waiting for the keyboard but this really does not change the procedure presented above, only the form is different.

It is clear then that in general, nesting conditional loops will require control of the multiplexer,both when we store the program counter into the stack to start the loop, and when we test the condition at the end of the loop.

4.5.- **Nesting Counting Loops**.

Counting loops cannot be nested with another counting loop, simply because the Am2910 has only one auxiliary register to perform the count. Naturally, one counting loop could be nested with several conditional loops in any order, without any problem. The limitation is that we can have only one loop using the register. Another possibility is to convert all of the counting loops minus one into conditional loops. This can be done externally in counters that can be loaded and decremented under the control of the micro code. The overflow of the counters are sent to the condition code multiplexer and the micro code selects the appropriate one at every step. This might be necessary for the design of controllers.

One type of loop that can be solved as a counting loop but that there are other types of solutions, is the timing loop. Many times, specially in controllers, it is necessary for the unit to wait for a certain time. This can be done with a timing loop, if the waiting time is within the range of the 12 bits of the register.

Another solution that does not require the use of the register, is to have a loadable counter under the control of the micro code. The counter can be loaded from the pipeline register and will decrement every clock. The overflow of the counter is connected to the CI input of the Am2910. In this way, once the counter has been loaded, the Am2910 remains executing the same instruction while the counter decrements (or increments). When the counter overflows, the Am2910 increments its program counter and the loop ends. Other arrangements of this type could be produced.

The three way loop is only a counting loop that ends in a three way branch and has three possible outcomes depending on the condition and on the counting. You should note that the end of the count ends the loop but that the outcome depends on the condition.

4.6.- **Alternatives**.

The problem we will face now is a typical engineering problem. We have studied the Am2910 and it seems to be a very nice device for a sequencer. It has many capabilities and variations and permits us a great flexibility in the design. The question is: Do we really need it?

To try to get an answer to this question we must consider all the angles. The Am2910 gives quite a lot of flexibility for the generation of loops, subroutines, etc. If we need this flexibility, then it is natural to use it. Compare this condition with the analysis of the fetch in a computer controller. In this case, the sequencer only produces the next address until it is known that the fetch has ended, when it loads the next address from the MAP read only memory into the program counter.

It is clear that these functions could be performed by a counter, say a 74LS193 to mention one, that has a clear input, an increment function and a load function. So, in a normal computer environment, especially of the reduced instruction set computer variety, the sequencer needs three instructions from the repertoire of the Am2910: JMPZ, CONT, JMAP. JMPZ means that the next value of the micro program counter is zero and it is executed by activating, with a bit from the micro code, the clear of the counter. Another bit of the micro code can be used to activate the count up of the counter, thus incrementing it and produce the next address in sequence, executing the instruction CONT. Finally, by using another bit of the micro code to activate the load function of the counter it is possible to execute the function JMAP, where the output of the MAP read only memory is loaded into the micro program counter.

So, in the case of a normal computer we could replace the Am2910 with a counter. In a computer we will need to implement the conditional operations. This requires external logic to develop the condition and, in the case of the counter, the external logic can include bits from the micro code to activate or deactivate the condition. In this way, when executing a conditional instruction, if the condition is satisfied the signal becomes a load and if the condition is not satisfied the signal becomes and increment.

What we need to study now is if there is any advantage of doing this in a practical design. First, we should consider the cost. The cost of the Am2910 is not that large and probably it will be little advantage, if any, in replacing it with a counter, the conditional logic, and probably a register to hold the return address. From this consideration alone, a complex set of conditions will tilt the balance in favor of the Am2910 and away from discrete logic.

Cost is not the only consideration of an engineering design. When designing computers, speed is a very important consideration. The delay in a Am2910 is something to be considered. A device cannot perform such complex function without some logical complexity. Although the Am2910 is built with very fast ECL logic inside (and a shell of TTL logic), the delay in the chip is on the order of 40 ns. with a

minimum clock period of 50 ns. Equivalent values for military devices are 46 and 51 ns. Considering the use of fast devices around the Am2910 we end up with a period for the clock in the order of 100 ns., minimum. This says that any system we build using the Am2910 has a maximum clock period of 10 Mhz, regardless of how fast the other devices can go.

The speed developed above will be computed in more detail later when we develop methods for time analysis but it matches the analysis made in the Advanced Micro Devices book, page 5-178. The final meaning of this number is that we cannot develop a system that runs faster than 10 micro instructions per micro second. Since a normal macro instruction in a computer takes from 6 to 10 micro instructions, including the fetch, this means we cannot develop a computer that runs faster than say, a million instructions per second.

At this moment we need to go back to the thoughts we had when we first started. We said "with so many good integrated computers in the market, why design another one?". The speed we got above is slower than the 68000 and much slower than the 68020. So, our device could be better implemented using a 68000 or 68020 as the case may be.

That is the reason why we mention the discrete logic solution. If this is our problem, we have of the shelf counters that work at 100 Mhz, with the addition of faster read only memory's that are coming to the market, faster logical devices, we could develop a system where the sequencer runs at 50 ns.

Note that everything we have been saying is the basic speed of the computer. Later we will apply pipelining, overlapping, factoring, and all the tricks in the book, and our 50 ns. could be lowered by a substantial factor that can be as large as 5. Incidentally, the problem solved in the Advanced Micro Devices book, page 5-178 is one that permits many of these tricks or speed up procedures to be used. Then, a computer that executes micro instructions at a rate of 10 or 20 Mhz is not illogical.

4.7.- **Other Devices**.

Advanced Micro Devices offers other sequencers with various characteristics: the Am2909 and Am2911 are four-bit cascadable sequencers; the Am29331 is a 16-bit sequencer; the Am 29540 is a special sequencer to produce the addresses for the FFT computation.

THE DESIGN OF COMPUTERS USING BIT-SLICE TECHNOLOGY

Michael E. Valdez, Ph.D., P.E.

SLICE #5

Slices: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9

THE ARITHMETIC LOGIC UNIT

Every computer needs some device or devices to perform operations on or with the data. The most common of these devices is what has become to be known as the arithmetic logic unit, that performs arithmetic operations (add, subtract), as well as logical operations (AND, OR, EXOR, shift, rotate). Other devices provide either the capabilities for performing multiply and divide in micro code, or provide these operations in hardware.

Naturally, the operations that can be performed with the data are not limited to mathematical manipulations and non-mathematical operations are becoming more common, like operations on strings of characters, filtering, etc.

5.1.- **The Arithmetic Logic Unit**.

The core of any computer is always the arithmetic logic unit, to the extent that Advanced Micro Devices calls microcomputers to all the bit-slices for arithmetic logic units. Let us start with the Am2901, which is the initial device. This device has gone through a series of modifications that have produced a better, faster

and cheaper device; the perfect modification. In this way we have the Am2901, Am2901A, Am2901B, and now the Am2901C.

The Am2901 is a complex device, it can be used for many applications. It is a 4-bit slice that can be cascaded to any length. The arithmetic logic unit in the Am2901 is an eight function unit. The function is selected through the instruction pins of the device.

Fig. 5-1 shows a block diagram of the Am2901 with its most important elements and pins. Please refer to this figure in the descriptions that follow.

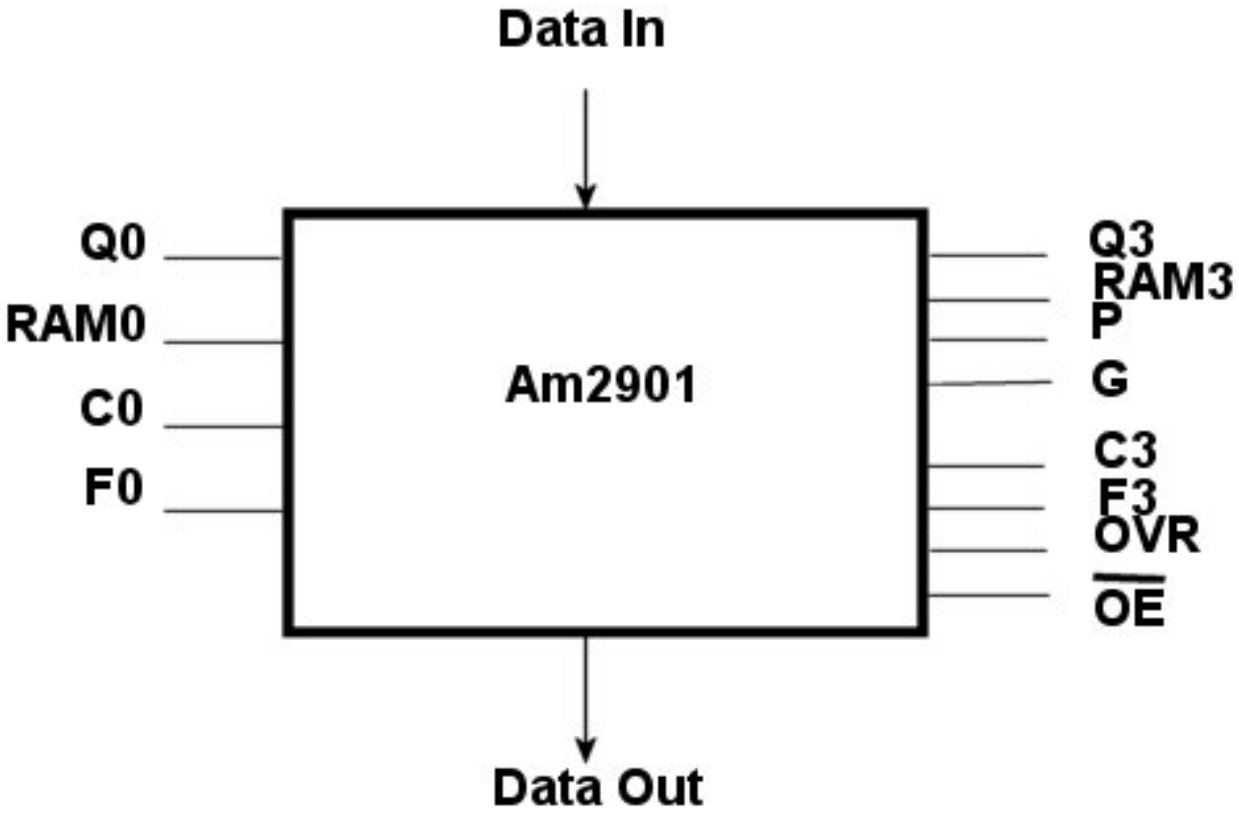


Fig 5-1

5.2.- **The Two-Port Memory**.

There is a block of 16 registers that is a two-port memory. This means that every memory location can be accessed through two ports. The block of registers has two memory address inputs and two data outputs. The content of the memory location addresses through address A comes out to data output port A, and the same with B. These ports are latched, so it is not necessary to hold the address but for one clock. The address inputs are 4-bit wide and all the devices are paralleled; the data is 4-bit wide and they are cascaded

from device to device. The two addresses can be equal or different without any restrictions. This is very useful to speed up the register to register operations that become a one cycle operations. The input to the memory is only through the B port. That is, the B port is always the destination register.e Q and/or in memory.

5.3.- **Shifts and Rolls**.

The Am2901 has shifters in the input of the memory. The shifter is only a three position multiplexer that in one position connects the bits in their proper location; in another position it connects each bit to the next lower bit location; and in the third position it connects each bit to the next higher location. So, by selecting the position of the multiplexer we can shift up or down one bit. The higher bit of each device goes to a pin to be connected to the next higher device and the same with the lower bit. This permit us to shift longer words. The Q register has also a shifter in the input. When we shift Q, we shift both Q and memory, but memory can be shifted alone.

More complex shifts require multiplexers outside the device. Imagine we want to roll; we need some way of connecting the higher bit out of the most significative device into the lower bit of the the least significative device. All the other higher bits are connected permanently to the lower bits of the next higher device. If we only need to roll in both directions, a permanent connection is enough because the pins are bidirectional. The problem is when we want to roll sometimes and sometimes to shift; when we shift we might want to put a zero or a one into the bit that is vacated. We may also want to put the content of the carry into this bit. Another possibility is to copy the sign bit, for arithmetic shifts. We need a multiplexer into the low input of the lower device and in the high input of the upper device. This change does not solve all the problems because we cannot shift or roll in the direction opposite of the multiplexer; so we need another multiplexer at each end. These two multiplexers provide for the different possibilities of rolls and shifts, arithmetic and logical shift, as well as different types of signed numbers. We need similar multiplexers for memory and the Q register.

The Am2901 gives all the necessary status bits that are connected to the Am2904 or a status register, for storage. The Am2904 provides for the possibility to preserve the status of the machine at the micro code level, saving the status at each micro code instruction, and the machine status at each instruction. The Am2904 permits us to load one or the other when we need it.

5.4.- **The Instruction Set**.

The easiest way to understand the instruction set of the Am2901 is to divide the instruction into three fields: The source of the operands, the operations, and the destination. Each one of these fields has three bits so there are eight choices for each one of them.

0	A and Q
1	A and B

2	0 and Q
3	0 and B
4	0 and A
5	D and A
6	D and Q
7	D and 0

where A and B are the outputs from ports A and B of the register block, Q is the auxiliary register, D is the input from the data bus, and 0 is the value zero applied to the input. The left value is connected to the input R and the right value to the input S of the arithmetic logic unit.

The operations provided in this device are selected by the second field, and are:

0	R Plus S
1	S Minus R
2	R Minus S
3	R OR S
4	R AND S
5	R* AND S
6	R EXOR S
7	R EXNOR S

where the * is used to indicate complement. Note that the source of operand does not permit all the choices; for example we only have connections from A to R and B to S. Other connections will be necessary only in the case of subtraction but, for subtraction we have R-S and S-R. So we are covered.

The eight choices for the destination can be divided into shifted and unshifted instructions. The instructions are:

0	F connected to register Q and Y
1	F is not stored and F connected to Y
2	F connected to register B and A to Y

3	F connected to register B and Y
4	F shifted down connected to B and Q, and F to Y
5	F shifted down connected to B, and F to Y
6	F shifted up connected to B and Q, and F to Y
7	F shifted up connected to B, and F to Y

where B is the register addressed by port B, Q is the auxiliary register, Y is the output to the data bus and F is the output of the arithmetic logic unit, the result of whatever operation was performed.

There is a pin for a signal that enables the output of the arithmetic logic unit and connects it to the data bus; this is important because the arithmetic logic unit is only one of the units that will be connected to the data bus.

5.5.- **Applications of the Arithmetic Logic Unit**.

In order to use any of the arithmetic logic unit it is necessary to have a number of control bits and/or multiplexers to route the signals that we need.

First, we need to produce the addresses for the registers that will be used in the operation, if any. This depends on the operation and on the application. In certain circumstances this addresses come both, or one, from the micro code, like in the case of internal operations required by the computer; in other circumstances they come from the instruction itself, like in the case of register to register instructions; in other circumstances we may have fixed values that are used very much and that are hard wired. In most cases we need multiplexers at the input of the addresses to the memory bank to select the proper source for the address, if not the address itself. Note that it is possible to have operations where no register is involved, like when we read a value first into Q and then we add this value to another coming from the data bus, like in a memory to memory operation.

The second point is the selection of the source. Most of the time this value comes from the micro code, but it is not necessary that this is always so. We could device a unit where this value is responsibility of the programmer and it comes from the instruction. This is done very seldom because it complicates the work of the programmer, who needs to understand the operation of the unit. In this case, we will need a multiplexer at the input of these bits, to be able to control the arithmetic logic unit from the instruction and from the micro code.

Then it comes the selection of the operation to be performed. It is more common here to leave it at the discretion of the programmer, to save micro code. There are two distinct possibilities: one is that all the arithmetic logic unit instructions have a special piece of micro code and then, the operation, and the whole arithmetic logic unit, is always controlled by the micro code. This calls for a long micro code because we need at least the eight operations to be independently coded. The other solution is to devote three bits of

the instruction to indicate the operation and have a multiplexer to route these three bits into the operation selection pins of the arithmetic logic unit. Naturally, the sources must also be controlled some how, which complicates the matter. Notice that the operation to be performed depends on the selection of inputs. Leaving whole flexibility to the programmer will make the computer very complex to use, as mentioned above. If we leave full flexibility to the programmer, he will probably curtail this flexibility when writing the operating system by using only a few of the options. The design can be simplified if the superfluous options are not presented to the programmer.

This problem naturally does not exist in the case of controllers where there is no programmer in the sense of user. The micro code contains all the necessary information and the user has only very limited selections.

The selection of the destination is also complex and in most cases will be done by the micro code. The comments presented above also apply to this input. Naturally, we could develop a computer where all inputs to the arithmetic logic unit come from the instruction. This will give full use of the hardware to the user but, as mentioned above, we force the user to learn how the hardware works and how to use it. Very seldom this is justified.

The arithmetic logic unit will also need a signal to control the output when we want to drive the bus. Note that in the description above, certain instruction put values in Y, when we say 'connected to Y'; this means only that the value is available at output Y, not that is connected to the bus. We need to explicitly enable the output of the device to connect Y to the bus.

All the other connections to the arithmetic logic unit are normally hard wired, like the interconnections to cascade the devices to a larger word, or through multiplexers, like when we wish to have roll and shifts.

In order to cascade several devices we need to connect every RAM0 pin to the RAM3 pin of the device below, every Q0 pin to the Q3 pin of the device below, we need to connect in parallel all the addresses to the memory bank, the instruction pins, the output enable.

The arithmetic logic unit also provides status bits that indicate the status after each operation. The sign bit is taken only from the pin F3 of the most significative device; the zero bit is taken by connecting together all the F=0 pins of all the devices, with a pull up resistor (these outputs are open collector). The carry and overflow outputs come from the most significative device. The carry output of each device is hard wired to the carry input to the next higher device. More about this in next paragraph.

5.6.- **The Ripple Carry and Carry Look Ahead**.

One of the problems of cascading arithmetic logic unit devices is that the output of the system depends of the total operation of all the devices, when arithmetic operations are performed. The reason is that during arithmetic operations the output of each bit depends not only of the inputs (the operands) but on the results of the operation with all the less significative bits. Imagine a 32 bit adder formed by cascading eight arithmetic logic unit devices. In order to get the result we need to wait for the least significative device to

produce its results, the carry of this device is applied to the next higher order, then we wait for this device to produce its output, and in this way until all the devices have produced valid output. This is what is called ripple carry, because the carry ripples through the devices until it get to the most significative one. Only then the result is valid. If we consider that the delay from memory address to carry output is 59 ns and that from carry input to carry output is 20 ns, the whole operation takes 59 + 7 * 20 = 199 ns.

When using large words, the time it takes to perform arithmetic operations with ripple carry is too long and the solution simple enough that should be used. The idea is to use the procedure of carry look ahead. It is possible to compute what the carry of a four bit operation is going to be, without waiting for the end of the operation. In a larger word, we divide the word in nibbles and compute the P (carry propagate bit) and the G (carry generate bit) and, by combining them we can generate the final carry, and all the intermediate ones, with very low delay, and while the other devices are computing the sum or difference. This is the function of the Am2902 we will study next.

5.7.- **The Am2902, Carry Look Ahead**.

The Am2902 is a small device that provides high speed carry look ahead for the Am2901 and similar devices. Fig. 5-2 shows the circuit diagram of this device. The Am2902 can handle up to four arithmetic logic units, or 16 bit data words with Am2901's.

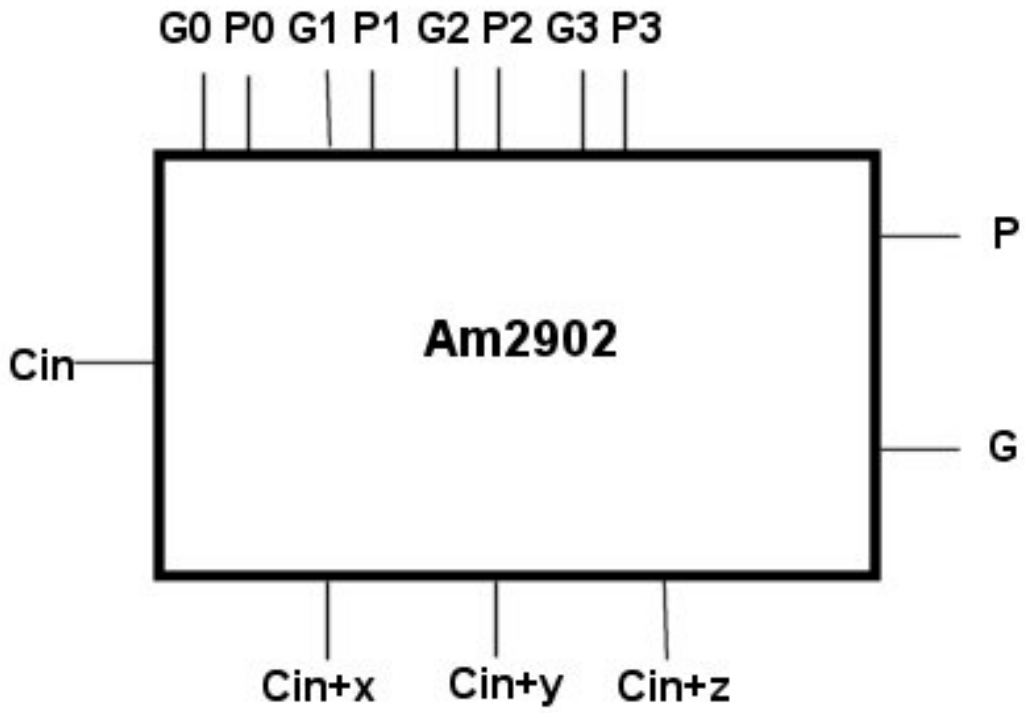


Fig 5-2

The principle behind the carry look ahead is that given the bits of an addition and the carry input to the least significative bit, it is possible to compute all the carries with two-level logic. Normally this is done in another way, to permit cascading of the slices.

Two signals are generated in each slice, the carry generate and the carry propagate. Consider the case of one-bit addition. If any of the bits of the addition is a one, and the other is a zero, then the carry input to the addition will propagate to the output. If both bits are a one, a carry will be generated at that stage, regardless of the value of the carry in. The equations to develop the carry generate and carry propagate signals for a four-bit stage can be derived similarly.

The Am2902 combines the P and G bits from all the 4-bit devices that are cascaded and it generates the intermediate carry input for all the devices, as well as the final carry for the operation. As mentioned before, the difference is time. The maximum delay in the Am2902 is on the order of 14 ns; the maximum delay in the Am2901 is on the order of 40 ns for the generation of the G and P signals, and we have to add 22 ns for the final stage. This gives us 76 ns for the whole operation with 16 bits.

In the case of 32-bit operations, we need to have one Am2902 for each half of the word (each 16 bits), and

another one to propagate the carry between the two, which gives us an additional 14 ns of delay, to a total of 90 ns, instead of 199 we had before. Naturally, the carry propagation between the two halves could be done by simple ripple, at a delay penalty.

This device does not need any control and, once connected into the circuit provides the carries without further consideration. As we saw above, the only consideration needed is during the design, to provide for sufficient delay for the operation.

5.8.- **The Am2904 Multi Function Device**.

The Am2904 is one of the many auxiliary device offered by Advanced Micro Devices. This one is an auxiliary device for the arithmetic logic unit. It can be used with any arithmetic logic unit device that does not have the functions provided by this device. It can also be used independently as a status register and condition code generation. A glance at the block diagram gives an idea of the many functions this device performs. The main functions can be classified as status preservation, condition code generation, carry generation, and shift/roll multiplexer.

This device provides two status registers that are intended to be used as micro status register and machine status register. In this use, the micro status register represents the status of the unit at each micro cycle, while the machine status register will represent the status of the unit at each machine instruction. Naturally, for these registers to represent anything, they must be loaded at the proper times.

Since this device has so many functions, controlling it is not that simple. Further more, a given combination of bits produces several different effects on the different parts of the device that all or some could be made active by a set of enable bits. The designer has to deal with a large number of bits, 17 in all, that he has to provide for the device to work properly. These bits can be divided into several groups. We will start with the instructions bits.

The instruction has thirteen bits that we can separate into three fields, bits 0 to 5 for the status registers, among other functions, bits 6 to 10 for the shift multiplexer, and bits 11 and 12 for the carry multiplexer. We start with bits 0 to 5, that can be written as two octal numbers. The meanings are as follows, breaking them in groups:

Micro Status Register

10 Clear the zero bit;
11 Set the zero bit;
12 Clear the carry bit;
13 Set the carry bit;
14 Clear the sign bit;
15 Set the sign bit;
16 Clear the overflow bit;
17 Set the overflow bit.

For the whole register

00 Move Machine to micro status register;
01 Set all bits of micro status register;
10 Swap the two status registers;
11 Clear all bit of micro status register.

Load the micro status register

06, or 07, Load register from the inputs I (that are normally connected to the status outputs from the arithmetic logic unit), ORing the old overflow with the new one to preserve the overflow bit;
30, 31, 50, 51, 70, or 71, Load the micro status register from the inputs I, but with the carry inverted;
04, 05, 20 to 27, 32 to 47, 52 to 67, 72 to 77, Load the micro status register from the inputs I, as they are.

Now, the machine status register:

00 Load machine status register from Y, which is a bidirectional set of pins that is normally connected to the data bus;
01 Set the machine status register;
02 Register swap as mentioned above;
03 Clear the machine status register;
05 Invert all bits of the machine status register.

Note that these operations are performed at the same time of the previous ones; for example, an instruction zero transfers the machine status register to the micro status register and loads the machine status register from the bus (Y). The same with all others. Naturally, we should be very careful when using this device in order to get the results we want.

Load operations with the machine status register:

04 Load for shift through overflow operation; that is, the zero and negative bits are loaded from the I input (which is normally connected to the arithmetic logic unit) and the overflow and carry bits are swapped;
10, 11, 30, 31, 50, 51, 70, or 71 produce a load with carry inverted; that is, all the bits are loaded from the I input except that the carry is inverted;
06, 07, 12 to 27, 32 to 47, 52 to 67, 72 to 77, load the machine status register from the I inputs.

Again, these operations are cumulative to those mentioned previously for the micro status register. Note also that the result of one of the instructions depend also of the enable bits and this is the way we control

the loading into one of the registers without affecting the other.

The operation of the Y output is controlled by one bit that is the output enable and by bits 5 and 4 of the instruction. If the output enable is high the output is high impedance independently of the value of the instruction bits; if bits 0 to 5 of the instruction are all low, then Y is an input regardless of the output enable bit; if only I5 is low, the micro status register is sent to Y; if I5 is high, there are two possibilities controlled by I4; if I4 is low the machine status register is sent to the output Y; if I4 is high, the input I is connected to the output Y.

The condition code output bit is controlled by all six bits of the instruction as described by Table 5.1 copied from the Advanced Micro Devices book. The different conditions for the shift multiplexer are displayed on Table 5.2 from the same source. Notice that I6 to I10 are devoted to this multiplexer.

Finally, the carry in multiplexer, whose output is connected to the carry in of the least significative device of the arithmetic logic unit is controlled as follows:

> If I12 is zero, bit 11 is copied into the carry output independently of any other bit of the instruction.
> If I12 is a one we have the following possibilities:
>> If I11 is a zero, the carry input is transferred to the carry output;
>> If I11 is a 1:
>>> If I5 is a zero, the carry from the micro status register is sent out, except when I3=1, I2=0 and I1=0 when the carry inverted is sent out;
>>> If I5 is a one, the carry from the machine status register is sent out except when I3=1, I2=0 and I1=0 when the carry inverted is sent out.

The Am2904 has the following enable pins, that must be supplied from the micro code, or hard wired as desired. Individual enable pins for each one of the four bits of the machine status register and one pin that enables all four bits. The operation is these pins is as follows: if the pin CEM is low, then the other bits control the loading of the corresponding status bits; if pin CEM is high, the machine status register cannot be changed, regardless of the value of the other pins. There is only one pin to enable or disable the micro status register. There is one pin SE to enable (when low) the shift multiplexer. There is one pin, OEY to enable when low, the input/output pins Y. There is one pin CEct that enables when low the output of the condition codes.

5.9.- **A Controller Example with the Am2901**.

In this section we will develop an example for the use of the Am2901 in a computer controller. We will assume that the system is as configured in Fig. 5-3, with an Am2910 as sequencer, an Am2901 as arithmetic logic unit. We will study the development of the micro code for this system to perform a definite function.

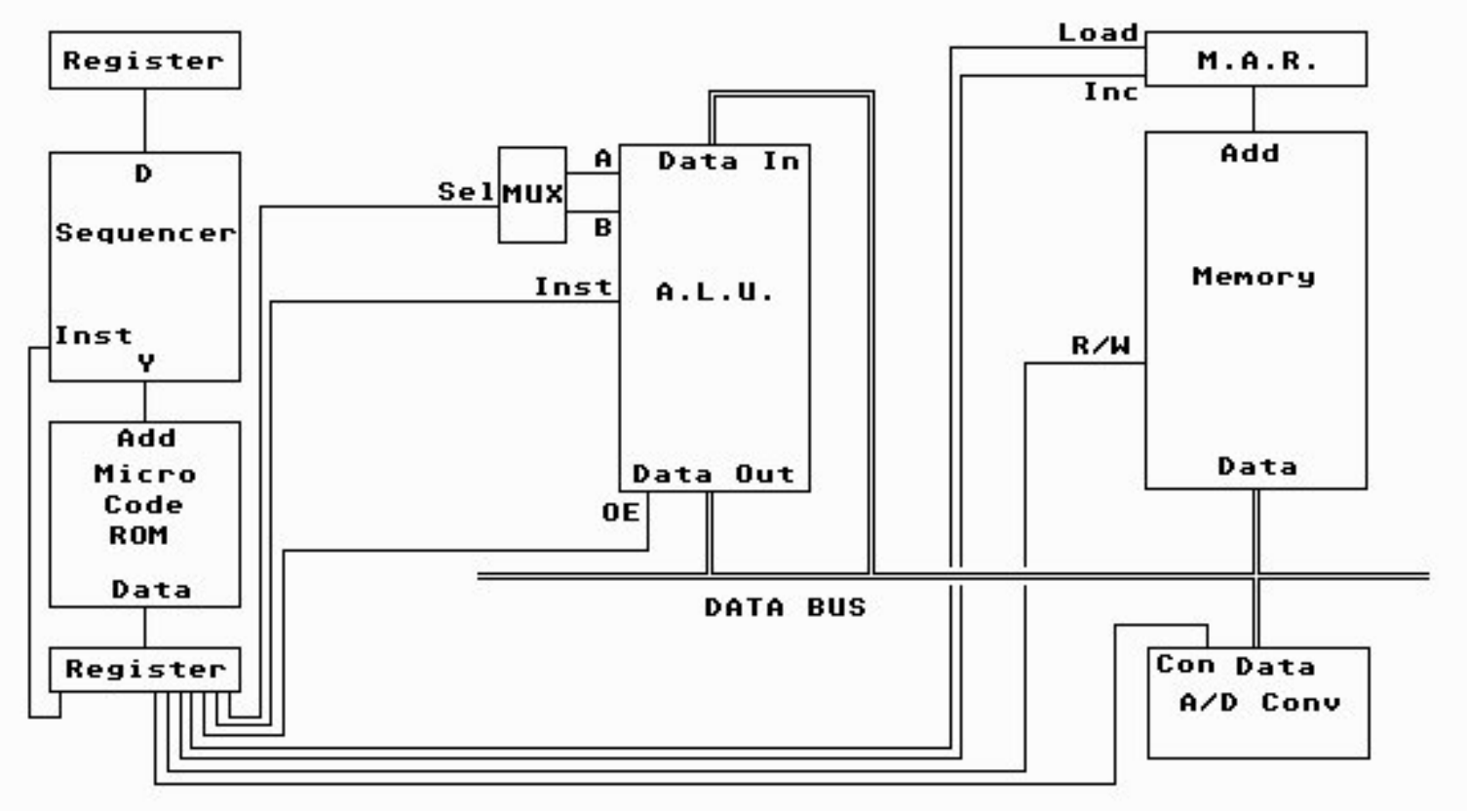


Fig 5-3

The problem we will consider will be that the device is receiving six-bit data values, from a analog to digital converter, for example. These values must be stored in successive memory locations and, at the same time, it is necessary to count the number of samples in the block, to terminate the operation, and to add the values so an average can be computed after the operation is completed.

One good way to start this type of work, is by developing the headings for the micro code. In the part of the circuit including the Am2910 we need the following: We need four bits for the instruction for the Am2910, twelve bits for the next address, if any, one bit to load the address in the memory address register, (we assume that the initial address for the memory address register is hardwired at the inputs of the register, recall this is a controller), one bit to increment this register, we need two bits for the read/write signal and the output enable to memory, one bit to enable the output of the digital to analog converter that also initiates the next conversion. So, this part of the micro code will look like:

#	Am2910		Memory Addr		Mem	D/A
	Inst	Adr	Load	Incr	R/W	Ena

For the part of the circuit that includes the Am2901 we assume that the register address has a multiplexer that, in one position puts the same hardwired address in both inputs and that is controlled by one bit of the micro code. We need nine bits of micro code to control the instruction to the Am2901 and one more bit for

controlling the output. This part of the micro code will look like this:

#	Reg	Am2901 Instruction			2901
	Mux	Sour	Oper	Dest	Out

Now, we follow the same procedure as before, analyzing the operations that are required and, as we do it, the micro code and any modification to the hardware will become evident.

Our operation starts by producing a dummy enable to the D/A converter, so it starts working; then, we need to load the initial address into the memory address register; finally, we need to initialize the register of the Am2910 to count the number of values to be received. Since our word is 16 bits long and the data is 6 bits long, we could receive up to 4000 values without concern for overflow. Since this value could be accommodated in the Am2910 register, we will use a counting loop as studied before. In cases of a larger number of samples and/or with more bits, we will need to provide for multiple precision addition and counting. We leave these variations as an exercise for the students. Note that in this case, it will be necessary to put the address of the registers from the micro code.

Going back to the development of the micro code, we note that all three operations can be produced simultaneously by the following line:

#	Am2910		Mem Addr		Mem	D/A	Reg	2901 Instruct			2901
	Inst	Adrs	Load	Incr	R/W	Ena	Mux	Sour	Oper	Dest	Out
	CONT	CONT		LOAD			E				

The next operation is to prepare for the loop by loading the Am2910 register counter. We will assume we have a register hardwired with the value to load. There is no reason why loading the register cannot be done at the same time as the previous operations, so we change the CONT for the Am2910 into a PUSH, push the program counter and load the register. Recall the conditional characteristic of this instruction. If no conditional instruction is required for the Am2910, this input could be hard wired; if not, it should be controlled from the micro code.

The loop is not too difficult. Enabling the digital to analog converter puts the data on the bus. Memory is commanded to write and, at the end of the cycle, the memory address register is incremented. At the same time, the value on the data bus is added to the register whose address is hardwired in the multiplexers and the result stored back into the same register.

At this moment it becomes clear that we need one more operation of initialization, that is to clear the register. Let us do this first. Clearing the register of the arithmetic logic unit can be done at the same time

as the rest of the initialization. In order to do this, we select the register multiplexer to H for hardwired, we select the source A,0, the operation is an AND and the destination is B. So, now the first line of our code looks like this:

#	Am2910		Mem Addr		Mem	D/A	Reg	2901 Instruct			2901
	Inst	Adr	Load	Incr	R/W	Ena	Mux	Sour	Oper	Dest	Out
	PUSH		LOAD			E	H	A,0	AND	B	

Note that we do not need to say anything about the memory R/W because the default state is read but output disabled; the memory address register does not need to increment and the output of the Am2901 should be disabled. Note also that we are performing four different operations in parallel, at the same time. This is one of the important advantages of designing a controller as a micro coded device, instead of a computer. Recall what was said in the previous slice with respect to speed. Although the speed of this controller might be similar to a computer, since we can perform many operations simultaneously, the performance is much larger. In a computer controlling this operation we will need to perform each initialization part as a separate string of instructions, that will be performed sequentially. Let us think about this, considering the 68000 mentioned before. The enable of the D/A converter requires a read operation; the loading of the pointer to the location when we want to store the data requires another instruction; the preparation for the loop requires another instruction to load into a data register the count we want to perform; clearing one of the data registers as accumulator requires another instruction; consequently, executing the initialization with a 68000 will require a minimum of four instructions.

After the initialization, we have the loop as explained above. This can be done with the second line of the code as shown below. The final line of the code is the end of the loop, that waits for the register to become zero. The micro code for this operation is then:

#	Am2910		Mem Addr		Mem	D/A	Reg	2901 Instruct			2901
	Inst	Adr	Load	Incr	R/W	Ena	Mux	Sour	Oper	Dest	Out
	PUSH		LOAD			E	H	A,0	AND	B	
	CONT			INCR	W	E	H	D,A	ADD	B	
	RFCT										

One interesting question that is left for the students to answer is if the last two instructions can be combined into one; that is, if the RFCT can be sent to the Am2910 at the same time the other operations are performed.

An additional point to consider is that the enable for the digital to analog converter is in general a pulse and, if we use the bit from the micro code, this will no be a pulse, especially if the change indicated above

is made. Then, it is necessary to AND (or OR) the bit of the micro code with a signal derived from the clock, to convert it into a pulse.

5.10.- **A Computer Example with the Am2901**.

We will study now developing the micro code for an example of a computer, some kind of a minimal computer using the Am2910 as micro program sequencer and the Am2901 as arithmetic logic unit, register bank, program counter, stack pointer, etc.

Fig. 5-4 shows the circuit diagram for this unit. Notice that we always need to analyze the specifications in order to develop a block diagram of the unit, which is the starting point of our development. Recall that we mention in the Introduction that one of the purposes of this text book is to develop an organized procedure for the design of computers.

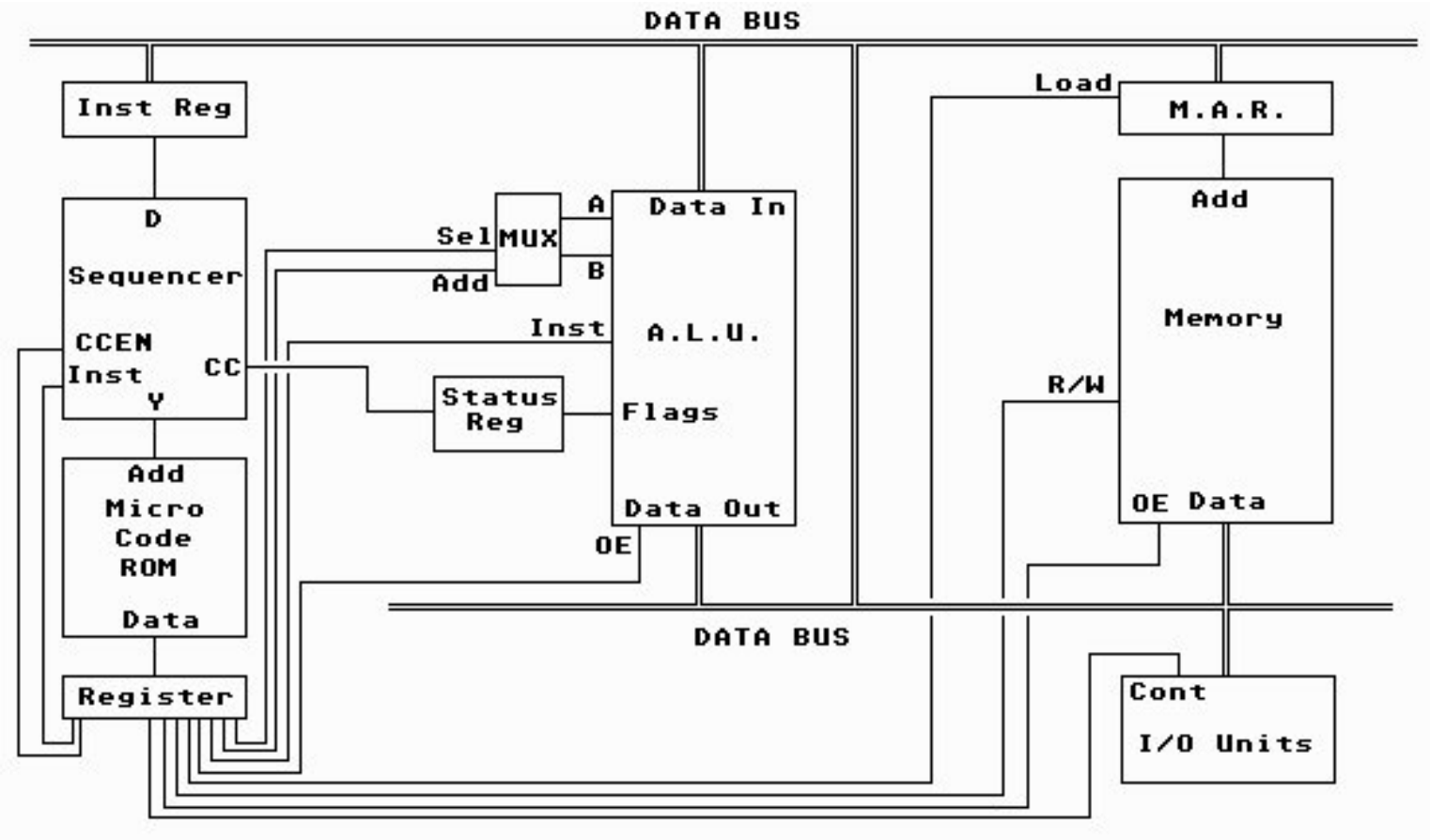


Fig 5-4

The second step of our design is to analyze the requirements and the block diagram, to come up with a set of instructions. A very common mistake at this point is to try to develop the whole set of instructions as one operation. This normally leads to a series of mistakes and, what is worst, that since the decision has been made of developing this set of instructions, the mistakes are cast in concrete and cannot be corrected. The best way to proceed to the design of a computer is, as mentioned above, to develop a block diagram and study the instructions one by one, adding them as it becomes evident that they are required.

The first logical step is to develop the micro code for the fetch and to be sure that the hardware can perform this operation. This will be the limit of this example and we will leave as an exercise for the students, the development of the other instructions they feel are necessary.

We have already developed micro code for other examples and the procedure we followed has given us good results; this procedure was to break the instruction into elementary steps, develop the heading for the micro code we consider we will need, develop the signals required for the execution of the steps without regard to optimization, and only then, go back and look at the whole thing and see where we can parallel operations, where we can overlap, pipeline, etc. As you gain experience in this enterprise, you will see the paralleling and overlapping as you develop the signals and merge these two steps to a certain extend. It is always wise to go back and analyze the whole code for fine tuning. This should be done as a minimum twice, once at the end of each instruction and second after all instructions have been developed.

We have looked at the fetch several times already, but this time we have different steps because of the hardware is different. The steps of the fetch are now:

1. Select the program counter address for both the address input A and B of the Am2901;
2. Select a 1 for the carry input of the Am2901;
3. Instruct the Am2901 to use A and 0 as source;
4. Instruct the Am2901 to perform an ADD;
5. Instruct the Am2901 to send input A to Y and store F in B;
6. Enable the output of the Am2901 into the data bus;
7. Instruct the memory address register to latch the data on the data bus;
8. Instruct memory to read;
9. Wait for memory to develop the data;
10. Enable the output from memory into the data bus;
11. Instruct the instruction register to latch.

Note that steps 1 to 6 not only take the program counter out of the Am2901 but also increment it and put it back in place. The rest of the steps are similar to what we have done before.

Now we need to think in the hardware. We need a multiplexer in the address input of the register bank so we can put the address of the program counter, and in other opportunities other values. Another possibility is to leave these address as always given from the micro code. Since we have already worked with multiplexers, this time we will use the micro code as the only source for addresses. The memory address register input must come from the data bus and all the other elements are as shown.

In the micro code we need the bits for the program for the Am2910, one bit to command the instruction register to latch, eight bits for the address of the registers of the Am2901, nine bits for the instruction, one for the output enable, and several for the carry input multiplexer. We need one bit to command the memory address register to latch, one more for memory read/write, and one for memory output enable. Then, our micro code have the following headings:

#	Am2910		IR	Am2901							Memory		
	Inst	Adr	Ld	RegA	RegB	Sour	Oper	Dest	OE	C	MAR	R/W	OE

As an exercise we will fill the micro code directly from the list of steps and later we do the optimization. We will assume the program counter is address 15 of the register bank; any one can be. So, our steps convert directly into micro code as follows:

#	Am2910		IR	Am2901							Memory		
	Inst	Adr	Ld	RegA	RegB	Sour	Oper	Dest	OE	C	MAR	R/W	OE
1				15	15								
2										1			
3						A,0							
4							ADD						
5								ANOP					
6									E				
7											L		
8												R	

At this moment we can analyze what we have. It should be clear that, in order to follow this simple minded approach the values that we have entered should remain enabled for all subsequent steps; for example, the 15 we put for register A address should be there for all eight steps. This points very clearly to the fact that all eight steps can be made concurrent, in a single step. With this idea in mind, we continue as follows:

#	Am2910		IR	Am2901							Memory		
	Inst	Adr	Ld	RegA	RegB	Sour	Oper	Dest	OE	C	MAR	R/W	OE
1	CONT			15	15	A,0	ADD	ANOP	E	1	L	R	
2	CONT											R	
3	CONT		lD									R	E

So, this is the micro code for the fetch in this type of architecture. During the development of the next

Slice we will have more to say about this architecture, where we will use it as an alternative.

5.11.- **Other Devices**.

There are many devices that can be used for arithmetic logic unit. Among them we have the Am2903 that has provisions for multiply and divide in micro code; the Am29116 that is a 16-bit arithmetic logic unit specially designed for controllers, the AM29332, a 32-bit arithmetic logic unit, and many others.

THE DESIGN OF COMPUTERS USING BIT-SLICE TECHNOLOGY

Michael E. Valdez, Ph.D., P.E.

SLICE #6

Slices: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9

THE PROGRAM CONTROLLER

When we studied the architecture of a computer, we mention we consider a computer a device formed by five parts: memory, input, output, processing unit, and control unit. This is the basic architecture of all computers in use today, that is called the von Newmann architecture, since he was the first to study it and describe it in the terms we use today. He did not developed this architecture since the same architecture has been used since the beginning of computers. This architecture has two other characteristics: it has a program stored in memory and it executes instructions sequentially, one instruction at a time.

6.1.- **The Program Controller**.

This Slice is devoted to the study of the part of the control unit that handles the program counter, the program controller. The program controller is that part of the control unit that takes care of the basic operations with the program counter, the routine incrementing, the change of the program counter when a jump is executed, saving it when a temporary change in the flow is effected, etc. In the program controller we need then, a program counter, a stack to save the program counter for subroutines or

interrupts, a stack pointer, means for incrementing the program counter, means of updating the program counter for jumps, means to add a displacement for relative branches, etc. The program controller might or might not include the memory address register, which is that register that drives the address bus for memory.

We have several possibilities for developing the program controller. It is possible to develop the program controller with separate logic, as an integrated circuit, as part of the arithmetic unit, and several variations of these possibilities. We will see that the difference between these solutions is more than detail. The decision on where to put the program counter affects very much the way the unit works, the instruction set, and the use of the computer.

6.2.- **A Program Controller with Discrete Logic**.

Let us study first, the development of the program controller as discrete logic, with the same memory and adder as the whole unit. Please refer to Fig. 6.1. Let us follow the functions that are necessary to perform and develop the hardware for the unit.

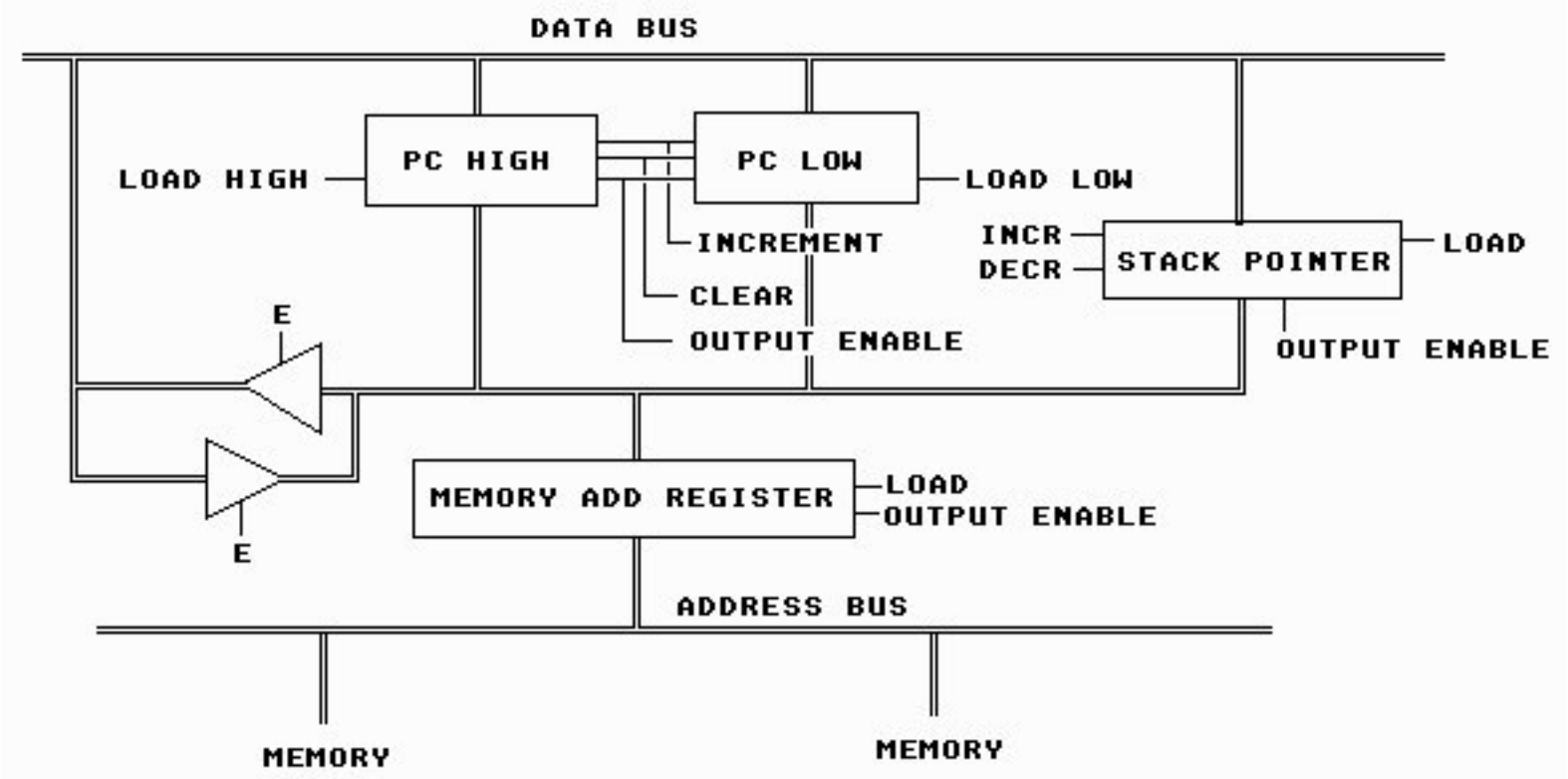


Fig 6-1

6.2.1.- **One Solution**.

The program counter can simply be a counter that must have the possibility of being loaded and of being incremented. We might also want the possibility of clearing the program counter for the reset of the unit. This could be done with a 74LS193, for example.

We need to load the program counter from the data bus, so the inputs are connected to the data bus. Note that this operation can be more that a simple load command. This happens when the address bus is wider than the data bus to have large memory spaces, a very common architecture today for medium size computers. In this case, it is necessary to load part of the program counter in one operation and then route the data bus to the other part of the program counter and load, in another operation. This requires that the inputs of upper and lower parts of the program counter will all be connected to the same bus, in parallel, and that two independent signals be used to load each one independently. This fact must be considered when studying the loading on the data bus. This fact also permits the use of pages, where the performance is increased when the old and new addresses are in the same page.

The program counter needs a path from its output into the memory address register so a fetch can be performed. It also needs a path to the data bus so it can be stored in the stack, and to the arithmetic logic unit so the program counter can be added to another value. Recall again the problem of the width of the buses.

The memory address register needs to drive the address bus. If there are other elements that can drive the address bus, like when we have direct memory access, we might need tri-state devices in the output of the memory address register. Note, though, that the memory address register could also be a counter and part of a direct memory access controller. More about this later.

The inputs of the memory address register should be connected to the data bus, with the corresponding problem of data width. Naturally, a separate bus could also be formed for the movement of the addresses. This still leaves the problem of the width of the adder.

The possibility of having only one path into the memory address register from the data bus, creates the problem that the data bus is used for every operation when an address is moved, requiring one or more additional cycles for all memory accesses. Since the memory access is always the slowest cycle in the computer, it is better to have an independent bus into the memory address register. This independent bus needs connections to the data bus. Please see Fig. 6.1 for a possible arrangement of this kind.

The other element that we need is some place where we save the program counter. We will assume first that the stack is in main memory. We need then a stack pointer, that is also a counter but this one needs not only a load and an increment but it requires a decrement function. This can also be done with a 74LS193. The output of the stack pointer must go to the memory address register and to the data bus so we can save the content of the stack pointer. The input of the stack pointer is only connected to the data bus. In order to use the stack pointer we need a path from the output of the program counter into the data bus.

It is possible to change the arrangement of the paths from tri-state buffers into multiplexers. This reduces the number of bits required for controlling the data paths, but it is slower because of the decoder included in the multiplexer.

6.2.2.- **An Improved Solution**.

There is another possibility that speeds up the computer by one or several cycles in every memory access. Please see Fig. 6.2. This possibility is to have the stack pointer, the program counter, the memory address register, and possibly the direct memory access controller, all driving the address bus. The advantage is that it is not necessary to move the content of the program counter or the stack pointer into the memory address register in order to initiate a memory access; it is only needed to enable the proper tri-state buffers or make the proper selection of the multiplexer. Recall that every register transfer operation requires one micro cycle.

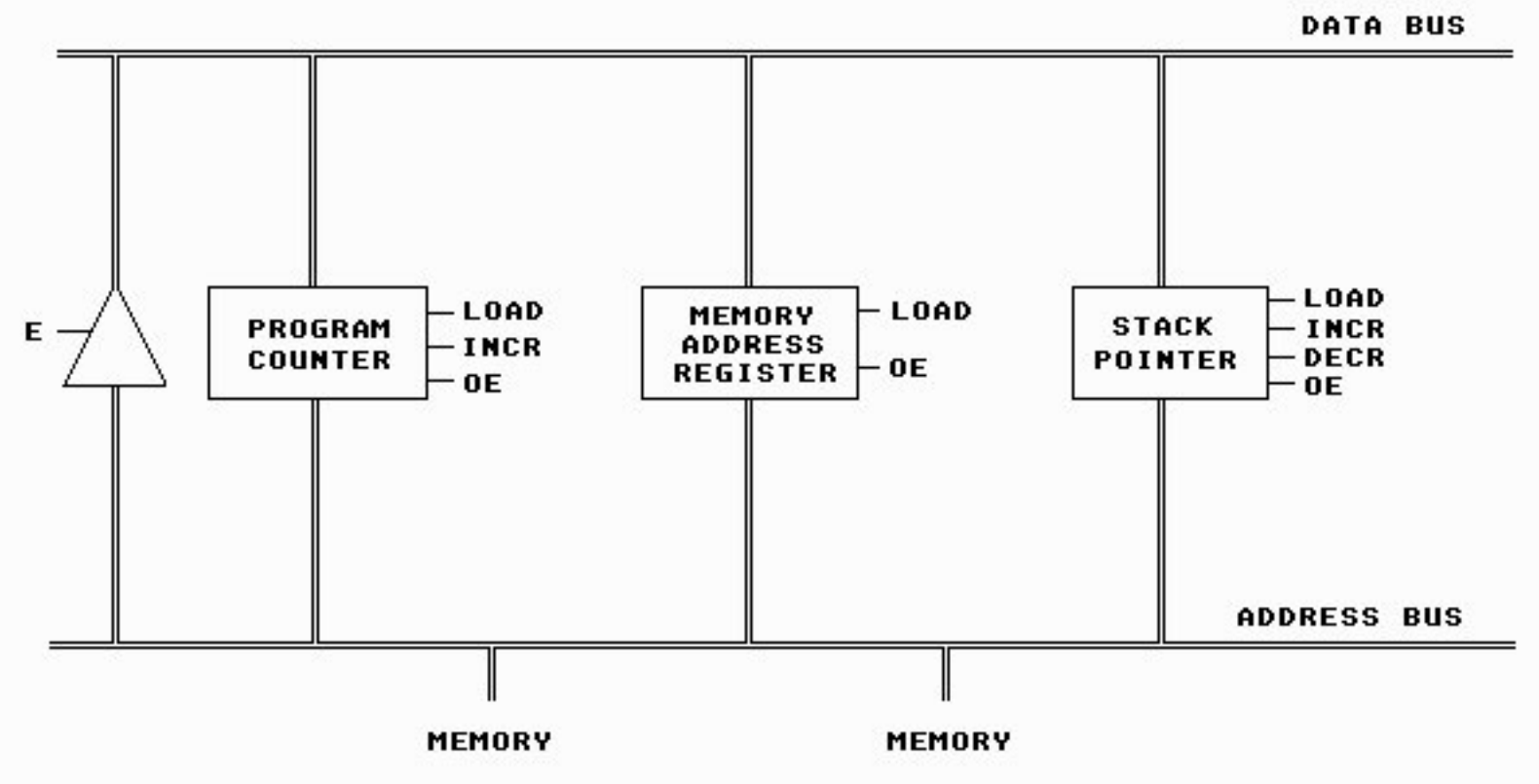


Fig 6-2

This arrangement does not prevent to increment or decrement the stack pointer or the program counter in the same cycle, because these devices decrement or increment at the end of the cycle. If we use this arrangement, these registers must always point to the desired address. For example, the stack pointer must point to the top of the stack, the program counter must point to the next address.

6.2.3.- **The Next Improvement**.

The next level of complexity in the design of the program controller is to put the memory for the stack as a separate unit, independent from the main memory of the computer. Please see Fig. 6.3. This arrangement has the advantage that we can perform a stack operation in parallel with memory operations. Since these two types of accesses are slow, paralleling them when possible, represents a substantial saving in time.

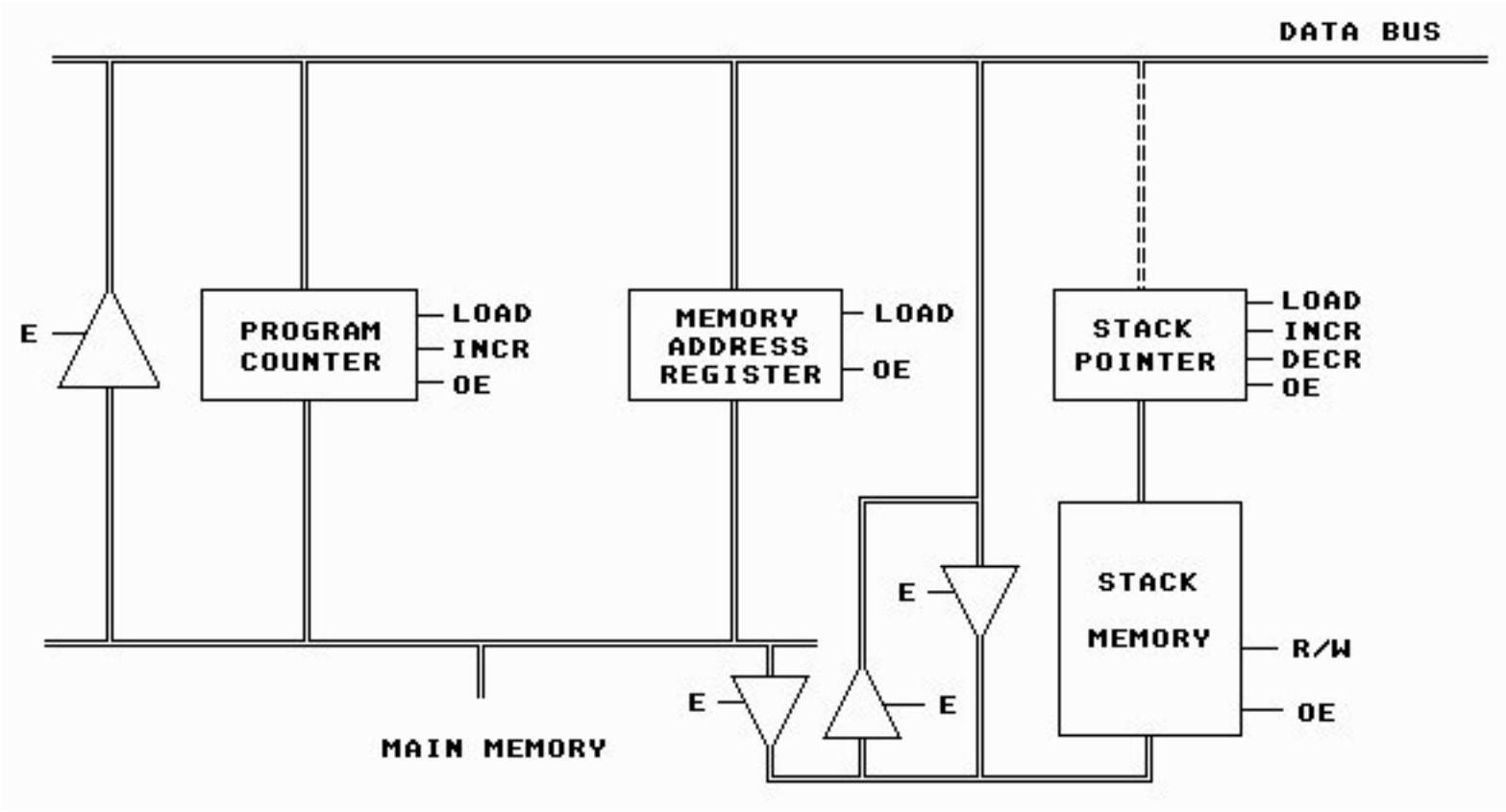


Fig 6-3

In this case, the stack pointer always drives the stack memory, there is no need or meaning for a path from the stack pointer to the memory address register. We need a control signal for the R/W line for the stack memory. All the control signals naturally come from the micro code. To use all the possibilities of the separate memory for the stack, we need to separate the internal bus from the external buses.

Notice that the parallel operations we are talking about have a very interesting characteristic. When we overlap the steps of the instructions we do it in a one by one basis and, in this way, we prevent the problem of pipelining that is the need to flush the pipe under certain circumstances. Notice that the flush of the pipe requires logic to make the decision to flush and wait for the execution of a certain instruction. In the case of overlapping, when we design the micro code we overlap steps without any possible problem during execution, and we gain maximum speed.

So the reason we bring the memory for the stack into a separate unit is that we want to execute steps in parallel without pipelining. So we need another bus that permits us to move the program counter to and from the stack. We need the output of the stack memory into the memory address register and into the data bus so we could move data around. We need paths into the stack memory from the program counter, from the data bus, possibly the stack pointer into the stack memory. Again, this arrangement could be done with multiplexers controlled from the control part of the micro code.

6.2.4.- **A Final Improvement**.

One problem of all these arrangements is that when it is necessary to perform arithmetic with the program counter, as when relative branches are needed, it is necessary to use the main arithmetic logic unit of the computer. This produces the next level of complexity in the program controller, that is to include a dedicated full adder for the program controller with the full width of the address. This, naturally, will permit more parallelism of the steps of any instruction. This full adder will need multiplexers in the inputs so we can route different pieces of data into the adder. Possible, the relative branches have the displacement as part of the instruction, in long word computers, or as a separate word, in short word computers, so we need paths to route part of the instruction (sign extended) to the full adder. The data bus should also be routed into the full adder. We will need paths from the program counter and, possible from the stack.

6.3.- **The Integrated Circuit Solution**.

What we have in the last solution is more or less a standard program controller with all the desired features. Naturally, there is an integrated circuit, the Am2930, which is a program controller with all these characteristics. This integrated circuit is a 4-bit slice with full adder, and multiplexers wherever we could need them. The Am2930 is cascadable to any size of address bus we might need, it has a very nice set of instructions, the stack is 17 words deep. There is a signal that indicates the stack is full and another that indicates the stack is empty, that can be routed to the status bits of the computer, so the programmer can test them.

The Am2930 has an incrementer for the program counter that is interesting when we need to prevent the unit from incrementing the program counter, as we did with the Am2910. It has an auxiliary register, in a sense similar to the Am2910, but that cannot be used as a counter. This register could be added to the program counter in various combinations. This register is basically used to store displacements.

The Am2930 is quite fast and has a large number of instructions, half of them conditional. It is interesting that the device is fast because the program counter is in the longest path of the computer. The instructions for the Am2930 are not the instructions of the computer but come from the micro code and these instructions are capabilities for the designer. Notice that the use of the Am2930 as program controller increases the size of what we call the micro program.

Something interesting about this device is the pin called instruction enable. The idea is that the same bits of the micro code that are used for the Am2930 might also be used for other purposes in the computer. The Am2930 works practically only at the beginning of the instruction, during the fetch and, at the most during the fetching of the addresses of the operands. All the rest of the instruction the device is idle. We can assure that it is idle by disabling it and, if we need, the control bits could be used in another part of the unit. Another possibility is to have different program controllers to switch the use of the computer between independent tasks. This will reduce the number of bits of the micro code.

Another interesting ability of the Am2930 is the instruction RESET, to initialize the device to a known state. This was not needed or possible for the Am2910 because we were there at the designer's level. Now

we are at the programmer's level and we can tell him what happens during the reset and where he must put his initializing routine. The instruction is zero. Naturally, this instruction is either zero or all ones that are the easiest functions to produce.

There is another unit, the Am2932, that is very similar to the Am2930 as a program controller. Any of these units can be used to control the program counter.

6.4.- **An Arithmetic Logic Unit as Program Controller**.

The other possibility we have is using an arithmetic logic unit as a program controller. When we have the stack in memory, the Am2901 can be used as a program controller, the 16 register could be used for storage of the program counter. Notice that this arithmetic logic unit should have the width of the address bus. One reason for using the Am2901 as program controller could be price. This dedicated Am2901 will be driving continuously the address bus.

Another possible use of the Am2901 as program controller will be in a limited multitasking environment, where the different program counters, stack pointers and/or status registers are kept in the different registers of the Am2901. Changing tasks requires only changing the meaning of the different registers, that could be done with a register at the input of the register address.

More interesting is the use of one of the registers from the arithmetic logic unit of the computer as the program counter and all the other 15 registers as general purpose register or stack pointers for different uses. This naturally requires that the address and data bus are the same width. Imagine the program counter is register 15, we use address 15 for both addresses A and B. Recall that there is an instruction that sends the content of the register pointed by A to the output from the device; at the same time, the content of A is sent to the arithmetic logic unit and it can be incremented and stored back in B, that has the same address as A. In this way we can send the program counter out to the memory address register and increment the program counter to have it ready for the next instruction, all in one micro cycle. Even in the case the output of the arithmetic logic unit drives directly the address bus, using the Am2901 as dedicated program controller, we could send the content of the program counter out, hold it in the output and at the same time increment it and store it back.

We have made our discussion with the Am2901 but any of the other arithmetic logic units in the market could be used as program controllers, some will be better suited than others and a careful analysis should be made to determine which to use.

6.5.- **Another Possibility**.

Another possibility we could study is using a sequencer as program controller. A sequencer normally has a register that can be used as a counter and, in this way, simplify the operation of certain instructions. A sequencer also has a stack that operates concurrently with the rest of the device. The Am2910 will only be good for small devices where a 12-bit address bus will be sufficient. Recall that there are other

cascadable sequencers with 4, 8 and 16 bits. In general, this does not seem as a very good idea, except in special cases. It is mentioned only because it might occur to the students.

6.6.- **Analysis**.

We will study now what consequences have the selection of each one of the solutions that have been proposed above.

The development of the program counter and all its accessories with discrete logic is the one that gives most flexibility to the designer but, it requires specific instructions for jump, branch with displacement, call to subroutine, indexing, etc.

Whether in the arithmetic logic unit or any other place in the computer, the control unit needs a block or registers. It is always possible to use one of these registers as program counter and another as a stack pointer, as was indicated above for the case where the registers are in the arithmetic logic unit.

The advantage of either of these uses is that we eliminate not only hardware but instructions because the normal jump instruction becomes only moving a value into one of the registers; that is, a MOVE immediate, that we need any way to load a registers with a value, produces a JUMP when the destination register is the program counter. We could even avoid the jump to subroutine and leave to the programmer to save the program counter where he wants. This permits simplifying the jump to subroutine in certain cases.

The main disadvantages are that a separate memory address register is needed and that nothing else can be done while manipulating the program counter. This is a consequence of using the data bus to move the program counter from the arithmetic logic unit to its destination.

The position of the program counter in the hardware determines then, not only the paths that are needed but also part of the instruction set. It is to be noted that the use of the arithmetic logic unit when the data and address buses are not the same width, does not seem to be advisable because of the two cycles required to move an address. Since the movement of the address is part of the memory access and, at the same time it blocks the data bus, this fact will reduce the performance of the computer. A balance between the advantage and disadvantages will determine which one is the path to take.

6.7.- **Using the Program Controller**.

At this moment, it should be obvious for the students how to use the program controller that is developed from discrete logic devices. We need to develop signals from the micro code for each one of the control signals that are required. For example, in Fig. 6.2 it is necessary to develop signals to load the program counter, to increment it, to enable its output into the address bus; the memory address register also requires signals to load it from the data bus, to enable its output into the address bus; the stack pointer needs signals to load it, to increment it, to decrement it, to enable its output into the address bus; finally,

the direct memory access controller will need signals to load the initial address from the data bus, to increment its value, to enable its output into the address bus. Recall that if the width of the two busses is not the same, it is necessary to have two signals for loading any of the registers from the data bus, the signals load low and load high. This fact also permits the use of pages, where the performance is increased when the old and new addresses are in the same page. This, naturally, must be included in the instruction repertoire to be used by the programmer.

The use of an arithmetic logic unit like the Am2901 for program controller should not pose any problem to the students, since it is similar to the other examples that have been developed for its use as an arithmetic logic unit.

The use of the Am2930 is also straight forward and will require only to become familiar with the instruction set and its capabilities. The use of an Am2930 in the multitasking environment described before can also be analyzed as two independent units, and the coding solved this way, one by one.

THE DESIGN OF COMPUTERS USING BIT-SLICE TECHNOLOGY

Michael E. Valdez, Ph.D., P.E.

SLICE #7

Slices: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9

OTHER DEVICES

This Slice is devoted to the analysis of other devices available from Advanced Micro Devices, as well as from other manufacturers.

7.1.- **Other Devices**.

Let us look now at what else is available. The Advanced Micro Devices manual provides information on other devices, like the Am2913, priority interrupt expander, the Am2914, vectored priority interrupt controller, the Am2925, clock generator, the Am2940, direct memory access address generator, the Am2942, programmable timer, counter, direct memory address controller. We will study now these devices.

7.1.1.- **The Interrupt Handling Devices**.

The Am2914 is a vectored interrupt controller that relieves totally the micro sequencer from the duties of

handling the interrupts. This controller receives the interrupt signals, analyzes the priority and disregard all calls that have lower priority than the one set by the sequencer. The only signals that must be accepted by the sequencer are those with a priority higher than the level of the mask. The Am2913 is an extender that converts each level of priority into a daisy chain. The Am2913 assigns a higher priority to the interrupting devices at the head on the chain.

The Am2914 receives interrupts from eight lines, but it can be stacked for large interrupt systems. It has an interrupt register, a mask register, a status register, and a vector hold register. The device is micro controllable and there are instructions to load and read the mask register, to clear and set the mask register, to clear and set bits of the mask register, to load and read the status register, to enable and disable interrupt request, to read the vector, four ways to clear the interrupts, and a master clear.

The Am2913 encodes eight lines into one of the lines of the Am2914. In this way, with the addition of eight Am2913's, the Am2914 can handle up to 64 interrupt lines. The process can be continued for larger systems.

7.1.2.- **The Clock Generator**.

One very important device in the Am2900 family is the clock generator. The Am2925 not only has the logic for generating the clock, but permits us to control the frequency, the duty cycle, etc.

The Am2925 produces stable operation from 1 to 31 Mhz. It permits to select with the micro code, cycle length from 3 to 10 periods of the basic oscillator frequency. There are also debounce circuits for a Run/Halt switch.

The Am2925 provides four different outputs for the clock, with different duty cycle and different position of the high with respect of the cycle. Output C1 is high for the first part of the clock period selected by the micro code, and low for the last fundamental frequency cycle; C2 is high at the beginning of the clock and it is low for the last two cycles of the fundamental frequency; C3 is also high at the beginning half of the clock and low for the lower half of the clock; C4 is low for the first period of the fundamental frequency and high for the rest of the clock. By combining these different wave forms, we can generate any wave form we want, especially, we can generate any transition we want. Note that when the period of the clock is changed, since these wave forms are constrained to integer periods of the fundamental frequency, some of these relationships are distorted.

7.1.3.- **Direct Memory Address**.

One of the most important ways of speeding up a computer is by performing certain operation by direct memory access. Input and output operations can be performed directly much faster than when the computer is in the loop. The reason is simple. In order to perform a block input, for example, the computer must read the input value into one of its registers and, in another cycle, move the value into memory. Direct memory access bypasses the cycle that moves the data into the computer. It is possible to

let the computer control the transfer, or have an independent unit to control the transfer. When the computer has control of the transfer, the computer has the duty to generate the addresses, signal the external unit, etc. In general, it is more convenient to have an external unit to control the transfer of data. The Am2940 and Am2942 are devices designed for this purpose.

The Am2940 generates the addresses, has a word count register that counts at each transfer, and it generates a signal when the transfer has been completed. The basic unit has an 8-bit address and it can be cascaded to any address length. This device has the capability of repeating the operation any number of times since the initial values are saved. The device provides four types of data transfer. We can have word count decrement, word count increment and compare with the final value, address compare and word count carry out. For each one of these cases, we can have the address register either incrementing or decrementing. The Am2940 accepts eight instructions from the micro code.

The Am2942 has a timer/counter and the direct memory address controller. It provides most of the functions of the Am2940 with the addition of two independent programmable counters. This device accepts 16 instructions, and it has an instruction enable pin that permits sharing the instruction field with other devices.

7.2.- **The Am29100 Family**.

The Am29100 family is a group of devices that have been developed by Advanced Micro Devices for the design of controllers; mostly intelligent controllers. It is assumed that the controller will have a sequencer, a micro code, and several other devices of the family to perform the required functions. The idea of an intelligent controller is that the controller performs by itself most of the functions that in regular systems are performed in software. By transforming complex software into hardware we gain first, speed in the operation of the computer and second, we free the computer to perform other tasks while the controller is working. This is one form of specialized parallelism. Notice that this type of parallelism does not have the programming problems of paralleling computers; then, it can be applied as much as desired. Typical applications are disk controller where the controller not only handles the disk, but it can handle scheduling several disks, ordering the accesses in the most efficient way, etc.

This family has, at this moment, the following members:

The **Am29112** is a high performance sequencer with an 8-bit address that can be cascaded for larger micro code read only memories. This sequencer has the advantage that is micro interruptable; that is, the sequencer accepts interrupts at the micro code level, totally transparent to the micro code. Recall that the Am2910 needs micro code to test and accept interrupts.
The **Am29114** is a real time interrupt controller that accepts eight interrupt lines and gives a priority vectored output, masking out those that are not needed.
The **Am29116** is a 16 bit three-port arithmetic logic unit with barrel shifter and a complex set of instructions.
The **Am29117** is a similar devices.

The **Am29118** is an 8-bit input output support for the Am29116, something like a port.
The **Am29141** is a fuse programmable controller, with 64 words of 32 bits of micro program memory.

7.3.- **The Am29300 Family**.

Now we enter into the new era. This family permit us to build very advanced computers without having to cascade many chips. These devices are 32-bits wide. The Am29323, a 32-bit multiplier which is very fast because you can multiply in one micro cycle. The Am29325, 32-bit floating point processor; the Am29331 16-bit sequencer, the Am29332 32-bit arithmetic logic unit, you can build a 32-bit computer with only one arithmetic logic unit chip, everything is inside. One chip replaces 8 Am2901 or Am2903 arithmetic logic unit's plus the carry look ahead chips, and all the auxiliary chips. We will save a lot of area on the board, time and labor for building.

This seems to be a good time to ask a question: How important labor is in a computer of this type? Converting eight chips into one reduces the labor cost, how important this is? We have to consider that we will not mass produce a computer of this type. It is not illogical to consider that a computer of this type can be a one of a kind, wire wrapped, or with a hand-made printed circuit board. So, the cost of labor affects the final cost of the computer more or less in the same way as the design cost. The design cost can be divided into a few units, and labor affects each unit.

The 32-bit arithmetic logic unit does not have registers so we have the Am29334 which is a dual port register file. Note that one of the limitations of the Am29116 is that it has a single port register file and it is necessary to manipulate the register address while the instruction is being executed. Another device is the dynamic memory controller, the Am29368.

7.4.- **The Am29500 Family**.

The Am29501 is a beginning of another series, the 500 series, which is devoted to digital signal processing. With this family we can develop co-processors that perform multiple functions; the Am29501 is an arithmetic logic unit unit with a multi port register file; the Am29509 is a 12x12 multiplier, the Am29510, Am29516, Am29517 are 16x16 multipliers, that give the 32-bit answer through a 16-bit output port.

Something interesting that you should keep in mind as solution to certain problems: Advanced Micro Devices calls these chips sine-cosine generators. We can mention specifically the Am29526, 27, 28, 29. What they have is simply very fast read only memories with look up tables that are good for the computers with very large memory spaces.

The look up tables can be used in computers with limited memory space by putting a register in the input of the read only memory where the computer writes the argument of the function; the computer then reads the same location to obtain the value of the function. This uses only one memory location. The

memory space required depends on the resolution that is required for the argument of the function. The reason why they have four devices is that the most significant part and the least significant part of the values of the sine or cosine are in different read only memories. If you need more precision that the one provided by one read only memory, you add the least significant part for more digits.

That is something you can even generate yourself. If you have a function that is too complex to be computed in real time, you can generate a look up table if you can do it with sufficient precision in the argument. Note that the precision in the value of the function requires more words, not more space. The memory space of the look up table is determined by the precision that is required in the argument of the function. A typical case is the computation of the Fast Fourier Transform where we need sines and cosines, but the resolution of the argument is not that great, you need the values for only a few arguments, so it is easy to make a table. If the memory space is prohibitive, it is always possible to use the single memory address solution presented above. The difference is time, the multiple memory address with the argument as memory, requires a single operation to read a value; the single memory address requires two operations. If we consider that in general, the argument must be loaded to a register prior to the access, the difference is not that large.

Going back to the available bit slice chips, we see that the series 500 is devoted more or less to digital signal processing; we have the Am29540, which is a Fast Fourier Transform address generator; this chip generates the successive addresses for the Fast Fourier Transform and it is not limited to the normal radix 2, but it handles other radices. Recall that the radix 2 requires that the number of data points is a power of 2. When you have several radices at your disposition, the number of data points needs only be divisible by any two of the radices that you have.

The Am29520 and 21 are multi level pipe line registers, required for this kind of work.

7.5.- **The Am2960 Family**.

Another family that is developing is the dynamic memory controller Am2960, a series of chips for memory management. They not only take care of the refresh of dynamic memory but the family includes the chips that are required for assuring data integrity. The problem is that as the memory becomes more and more compact the probability of an error because of the cosmic rays is increasing. When a cosmic ray or a secondary collides with the memory, it destroys the data. When using 16 K memory chips, the probability of data damage because of cosmic radiation is one failure every million hours of operation, which is not too large; using 64 K memory chips the probability of error becomes five failures per million hours and, as the memory becomes more and more compact, this probability increases very much. Notice that one error every hour represents a failure every 3-4 billion memory cycles. At 256 K per chip, the probability becomes sufficiently large that we need some way to maintain the integrity of the data. Since there are other errors that can also affect the memory, there is a whole family of chips that not only detects such errors but it corrects the first error, corrects most of the second errors, and detects any number of errors in the same word.

7.6.- **The Am29800 Family**.

This family has several members, like the Am29803, Am29806, Am29809, Am29818, Am29821 to 28, Am29833 and 34, Am29841 to 46, Am29853 and 54, Am29861to 64.

The purpose of this family is to permit the design of self-testing devices; that is, devices that can determine if they are ready to work. Naturally, this requires special devices, mainly for the registers that must be double, with the shadow register being a shift register.

7.7.- **The Motorola Family**.

Motorola has started to introduce to the market a series of bit slice devices built with emitter coupled logic technology, thus very fast. They also have translators from ECL to TTL levels.

Some of the devices offered by Motorola are the MC10900, an arithmetic logic unit slice, MC10901 an 8x8 multiplier, the MC10902 which is another arithmetic logic unit slice, the MC10904 a sequencer, the MC 10905 error detect and correct device, the MC10800 a 4-bit arithmetic logic unit, the MC10801 micro program controller, the MC10802 timing function generator, the MC10803 memory interface, the MC10806 dual access memory stack, the MC10808 16-bit shifter.

7.8.- **Some Comments**.

You see what is happening is that the advancement in integrated circuit technology not only increases the speed of the devices but it shrinks them, making it possible to put more functions in the same chip. Every advancement reduces the size of the lines which permits to put more elements per chip. So the question that comes to mind is if it is wise to put separate chips per function or put more functions in a chip; the other approach that is being taken is to go to the byte-slice or 8 bits in a chip, or even more. We have seen what seems like the extreme today, with slices that are 32 bits wide.

THE DESIGN OF COMPUTERS USING BIT-SLICE TECHNOLOGY

Michael E. Valdez, Ph.D., P.E.

SLICE #8

Slices: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9

TIMING

It was stated before, in the Introduction, that this textbook has two main tasks, beside the obvious one of teaching bit slice technology. The first of these two tasks is to develop an organized procedure for designing computers and computer controlled devices, which we consider has been achieved. The second task is to develop an organized procedure for the computation of the clock cycle time, and its optimization. This Slice is devoted to this task.

8.1.- **The Importance of Timing**.

It is very difficult to say too much about the importance of timing in computer design. Timing in the basic tool we have to determine the clock cycle of our computer or computer controlled device. If the clock period is longer than needed, our system will be slower than what it can be; if our clock is shorter than the minimum needed, there will be instructions or steps that will not execute properly, at least under special circumstances and it will be very difficult to determine what is happening, those mysterious faults that nobody can find.

The other side of the coin is to consider that the computation of the clock period, unless it is done by "intuition", is a tedious and long process. Since we have to spend that time doing those computations, we might as well do them right and get the right results.

It is important to realize that the clock period has to be longer than the maximum delay in any parallel path used during the performance of the operation under study, and under the worst possible circumstances. It

is important to consider that certain devices have different delay when the inputs change from a zero to a one, than when the change is in the opposite direction. Under certain circumstances, we know the direction of the change and we can use the proper value. If we do not know the direction of the change, as when we are switching data, we must take the longest delay. A delay shorter than this will produce those mysterious faults we were talking about before.

The next consideration is with respect to which specifications to use. The manufacturer some times gives minimum, typical and maximum values. In other cases the manufacturer gives only one value and "derating" figures. In other cases the manufacturer give firm values that they are willing to guarantee.

In the last case mentioned above, there is no problem with the use of the values, especially if the manufacturer of the devices you will use is a reputed manufacturer. If you are not sure of the quality of the manufacturer or when you do not know whose devices will be used, the best approach is to consider the typical values and, at the end, include a safety factor that will depend on the number of devices you are averaging and your judgement of the possible quality you will obtain.

8.2.- **The Ideas of Timing and Loop**.

We will call a loop any path, not necessarily closed, that starts in a register and that ends in a register. Let us analyze one such loop. We will consider the loop that starts in the memory address register, goes through memory, and ends up in the instruction register. The typical breakdown of this path is: -Delay from the output enable until the output pins of the memory address register are stable; -Delay from the memory address until the data is stable at the output pins of the memory chip; -Delay that it is necessary from the moment the input pins of the instruction register are stable, until we can clock the data.

Although this breakdown is correct, we need to consider other paths that might influence the outcome:

- Delay in the address decoders, if any;
- Delay from the chip select pin until data is stable in the output pins;
- Delay from the read/write pin until data is stable in the output pins;
- Delay from the output enable until the data is stable in the output pins;
- Delay in any logic that feeds the signals to the output enable of the memory address register, the read/write signal, the output enable of the memory chips, and any other signal used in the decoding equation of memory;
- Delay from the clock edge until the signals are stable in the pipe line register of the micro code;
- Any other delay affecting the signals.

As can be seen in the previous analysis, we should not limit ourselves to the obvious path of the signals, we have to consider all the signals that play any part in the operation to be performed. In the case we are studying, the path really does not start in the memory address register but it start in the pipe line register of the micro code, at the moment this register receives the command to latch the next line of micro code from the read only memory. This example shows that there are no-obvious parts of the path that have influence on how long the period of the clock must be.

Another aspect we need to consider is that our main path has many no-obvious paths that, in general, parallel the main path or one another. In this case we need to develop a diagram with all these path, analyzing the delays and how they affect the final results.

Let us follow in more detail the example mentioned above. Please see Fig. 8.1 for a complete circuit of this path and all its auxiliary paths. Fig. 8.2 shows an hypothetical timing analysis made for the circuit. We are assuming that the delay from address, read/write signal, chip select, and output select, to data output stable are different. We are assuming that the decoding equation for memory includes a signal from the micro code that goes through a multiplexer that has some delay, and that the pipe line register need some time. All these delays are shown qualitatively on the figure.

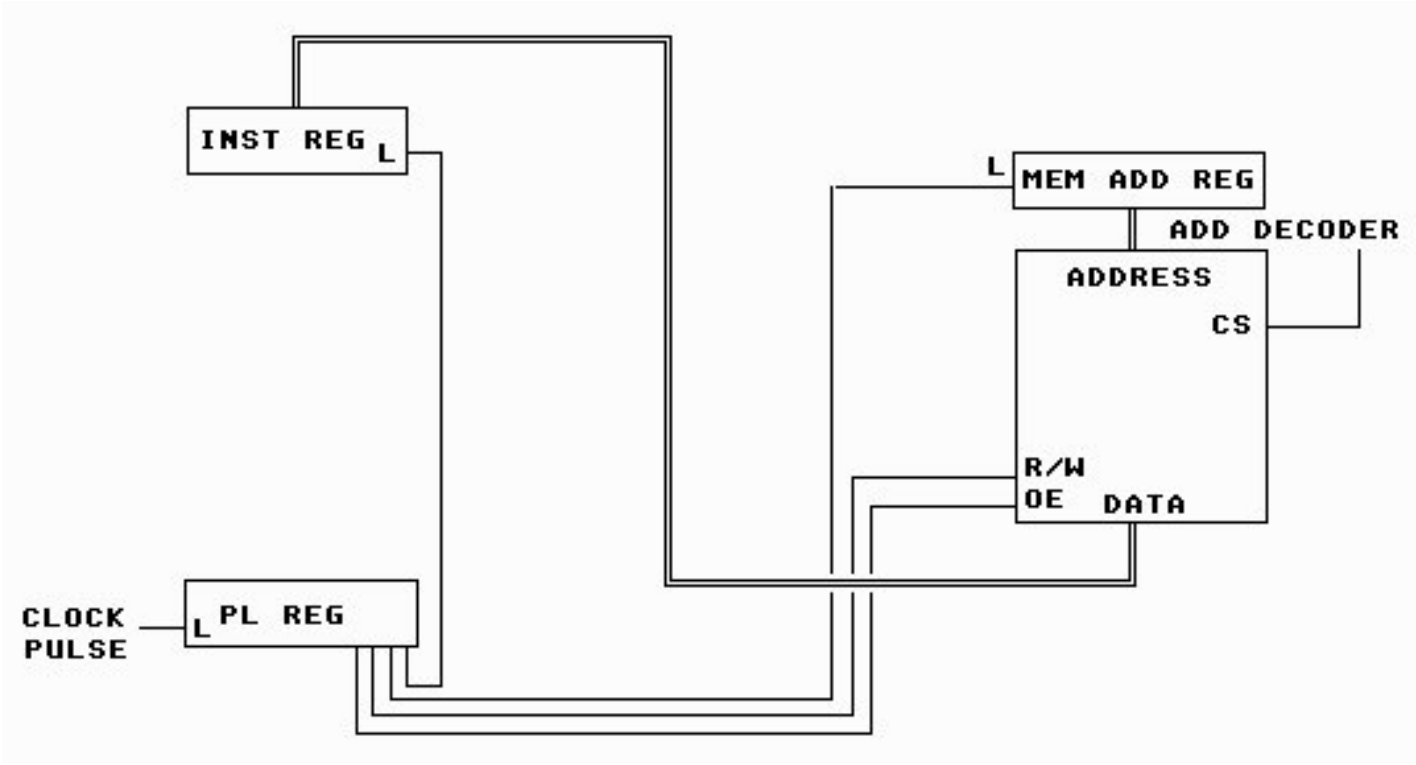


Fig 8-1

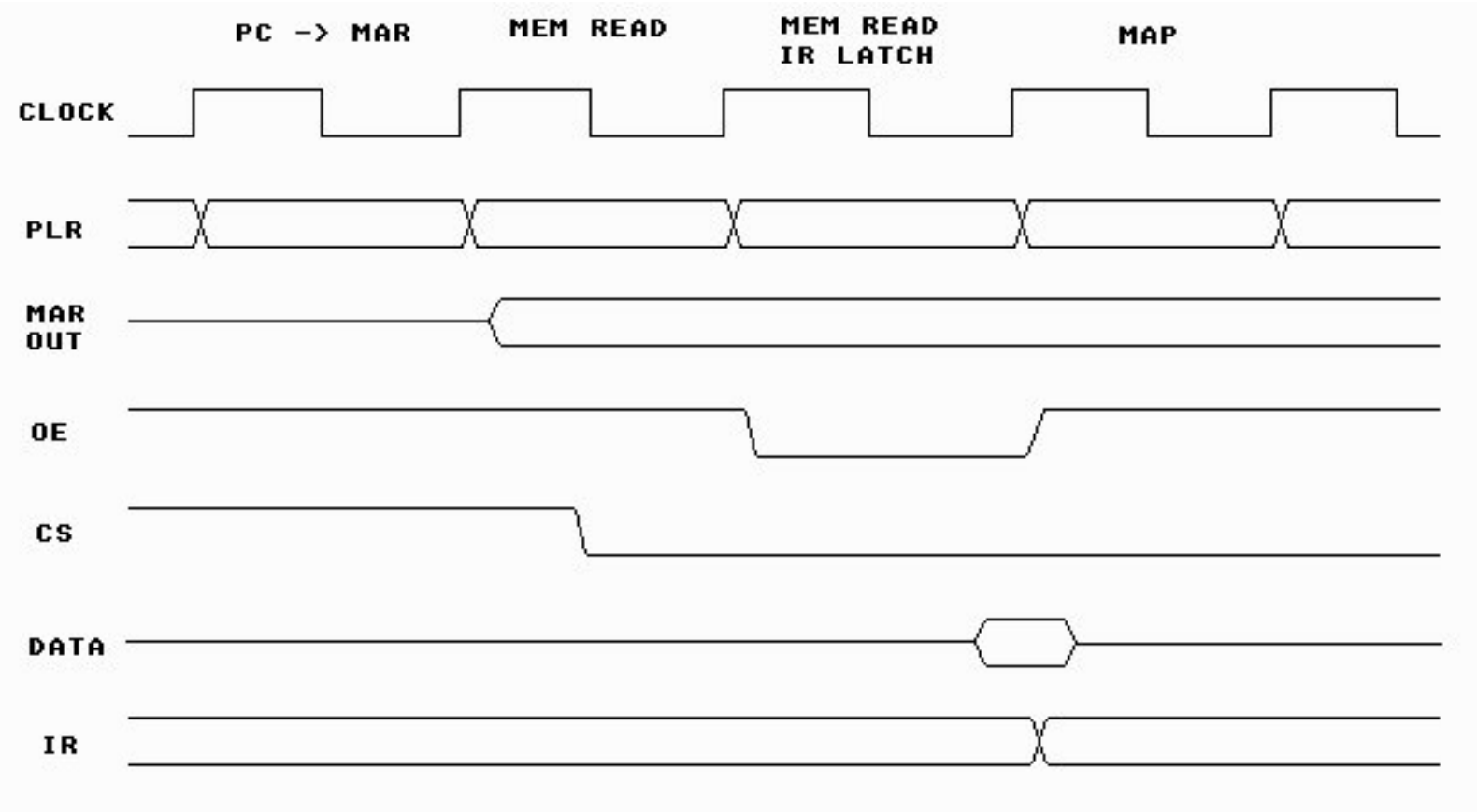


Fig 8-2

Notice that since the output enable signal is generated directly from the micro code through the pipe line register, it does not affect the operation because those delays are much shorter than all the others. The same is true, for this case, for the read/write signal. Note also that although the delay from the memory address register, through the memory, including the delay in the pipe line register, can be satisfied by a short clock period, a longer period is needed to satisfy the path through the chip select. This path starts in the pile line register, goes through the multiplexer of the extra signal, then through the decoder, at which time the other bits of the address are stable and do not affect the outcome; note that when the chip select is stable, the read/write line has been stable for some time, consequently it does not affect the operation. Then we need to consider the delay from the chip select to output stable, at which time the output enable is stable, and we continue from there through the previously mentioned path.

Note then, that the real path goes from pipe line register, through multiplexer, decoder, memory, to end in the instruction register.

8.3.- **Timing Analysis of the Paths**.

The main subject of this Slice is the computation of time delays through the different paths of the computer in order to define the period of the clock. The fundamental problem is how we organize our clock. The period of the clock has to be long enough so data propagates through the longest path. This is one possibility, that we call a single clock. The problem is to find the longest path and this can only be done by going through all the paths one by one, until the longest path is determined. The other possibility is multiple clock periods. The idea is the period of the clock assigned to each step is only long enough for

the propagation of the signals used in the step. This needs clarification because signals propagate through all the paths at every step. The point is two fold: imagine a step where memory is not used, and we have determined this is the longest path; then, if our clock is shorter than what is needed by memory, this unit will not finish its operation when the clock changes but, since memory is not being used, we don't care what happens in that unit.

The other side of this coin is when there has not been any change in the data feeding a path. Consider that the propagation of the data from the instruction register to all the different points where it goes is a very long path. This will affect the first cycle of all the instructions but, for the second cycle, this propagation delay does not affect our operation because data does not change in the instruction register until the end of the execution of the instruction.

The problem with the multiple clock is that we end up with too many clocks; one way out is to look at the Am2925, which is a clock generator. This device produces eight different periods from 3 to 10 periods of the main crystal. The clock period for any step is selected by three pins of the chip, that can be connected to three pins of the microcode. In this way, each step selects its own period and we limit the number of different period to the eight that the chip produces. Another point is that the maximum frequency of operation of this chip is 42 Mhz which permits us to produce periods from 71 to 238 ns. The 71 ns is normally too low for a clock, as we will see in a moment, so we can use a lower frequency and have a larger choice, more flexibility. The Am2925 produces four different wave forms that can be used for triggering devices at different points in the operation of the cycle. Typical case is memory address register, where the delay from wherever the address comes is very low and that it can be latched in the memory address register say 1/4 into the cycle and leave the rest of the cycle for the propagation through memory. That is, this device permits to divide the cycle in different pieces, using the time delay as best as possible.

Another possibility that we have given up in our design is to have an asynchronous system. This requires a response signal from each element to indicate when the unit has finished its duty. We use these responses to initiate the next cycle, clocking the Am2910, advancing a counter, or any other convenient way.

Something else to consider is to lower the frequency of the clock so we can have a single cycle for memory access. Note that if we do this we run into problems when trying to advance the fetch of the next instruction. It seems to be better the way we planed it, having a single clock period and two cycles for the memory access and wherever we can, we insert them during the execution of an instruction. Note that this is not pipelining but overlapping the instructions. The distinction is clear when we consider a branch instruction. Pipelining is done automatically but we will never include overlapping until we have defined the instruction to be executed next.

The important point at this moment is, that in order to have a solid base to make all these decisions, we need to perform the timing analysis of our circuit.

Now, how do we perform the timing analysis. Since we have to perform the analysis of all the paths, we can start any place. Let us start by analyzing the instruction cycle of the Am2910.

This analysis is normally done by getting several copies of the diagram and marking on them the different paths. The first thing we have to identify is from where to where the path goes. By definition, a path always starts in the clock input of a register and ends in the clock input of a register; that is, the time delay we consider involves from the moment data is stored or latched in one register until it can be latched in the destination register. The path under consideration starts when the clock latches the data from the microcode ROM into the pipeline register; there is a setting time for the pipeline register to its output; then we have the propagation from the instruction input to the Y output of the Am2910; then it goes to the address input of the microcode ROM and propagates to the data outputs, and finally has to settle before the pipeline register can latch the next instruction.

This requires:

From CP to pipeline register outputs
From I to Y in the Am2910
ROM address to data
Settling time for the pipeline register.

From the data sheets of the different devices we obtain, for commercial operating range we get a combinatorial delay from the program counter to Y of 70 ns for all instructions that do not use the counter register. Note that this delay starts with the clock pulse and we do not need to consider the delay in the pipeline register. Note how reading the specifications of a device changes the interpretation of the problem.

Looking at the data for the ROM's that AMD produces, we see devices from 25 to 75 nsec. We select one with 50 ns delay that seems to be the average value. The settling time for the pipeline register, assuming we use a 25LS07 register, is given as 5 ns. Note that the 25S07 has only 3 ns of setting time. So, the total delay for this path is 125 ns.

The execution of the instruction jump to map might have a longer delay because we have to recognize the instruction and pass D to Y. Note that we can eliminate the MAP ROM to reduce the length of the delay in this path.

Under certain circumstances it might be interesting to remove the MAP ROM and force the lower bits of the address to zeros; two zero bits leave four locations for the microcode for each instruction; three bits leave eight locations per instruction. There is a trade off here: on one side you have the fact that you save the MAP ROM but you loose in the microcode ROM in spaces that are not used for the short instructions. On the other hand, removing the MAP ROM reduces the time delay from the instruction register to the pipeline register. Since this is probably the longest path delay that will set the period of the clock, reducing this delay speeds up the computer by the delay of one ROM.

Let us now analyze one of the memory paths, starting with the fetch. In this case we have the delay from the program counter, through the multiplexer, through memory, to the instruction register. The first point to analyze is if at the moment of the fetch the program counter has already finished its incrementing. Note

that the minimum time for any instruction, even those branches that fail, there is one cycle after the instruction register has been latched and the program counter is ordered to increment. We can assume that during this cycle the program counter has incremented and it is ready. Note that this is not true for the case in which we have a two word instruction where in the first cycle of the operation of the instruction we need to put the program counter out to get the content of the next memory location. In this case the program counter has not finished incrementing; actually, at the beginning of the cycle the program counter starts incrementing and we have to consider this time.

In the case of the fetch we need the delay in the multiplexer, the delay in memory, and in latching the instruction register. For this last value we have 5 ns. For a multiplexer we can use the Am2932. The delay from the select to Y is 12 ns. Now, we can take 150 ns for the memory which is standard today for inexpensive memories. We have a total delay of 167 ns. So, our assumption of two cycles for the memory cycle was bad but, there is no way to know that unless we do these computations. At this moment, it seems that we can have a single cycle to cover the memory, since the delay of the Am2910 is in every cycle, it seems difficult to get shorter than 125 ns. Note that to shorten the memory path to 125 ns will require the use of 100 ns memory, which is not too expensive today and will be cheaper soon. We are considering static memory that does not require the additional delays in the refresh circuitry. Note also that we are not allowing for the delay in the decoder for the memory or any buffering that might be needed.

Another point we have to consider is the use of the more modern Am2910A or the Am2910-1. We have been using the old Am2910. With the Am2910-1 the delay is 75 ns. instead of 100 nsec we have considered. The cycle with the new chips are 105 nsec for the Am2910-1 path and 167 for 150 ns memory. Note that there is another memory access that is longer that the one we have consider.

Note that since our microcode does not have conditional micro instructions, like in multiply for example, we do not need to consider the propagation delay from the status register through the condition code multiplexer, to the Am2910.

Let us consider a memory access that will be longer, because we might need to wait for the program counter to increment. In this case we have something interesting and it is that the multiplexer will be open when the program counter has settle so we need to consider only the delay from the data to the outputs, not from the select to the outputs. After this, we need to consider the delay in memory and the delay in the arithmetic logic unit to latch the data. Note that the arithmetic logic unit will process the data after the memory has produced the data. The cycle cannot end until we give time to the status register to latch the new status of the machine, as a consequence of the transfer.

Another path that is very delicate is the conditional instruction. After the instruction register has been latched, we have a propagation through the select lines of the multiplexer, the CC input of the Am2910 to the output of the Am2910, or whatever execution it takes. This becomes very critical when we determine not to have a MAP ROM.

8.4.- **Timing Analysis of the Instructions**.

There is another way in which we can achieve the same result: in this method we do not work with the circuit diagram alone but we work with the instructions and we analyze how long it takes to perform each of the steps. Although, this is done with the circuit diagram, we are not looking from the point of view of the circuit diagram but from the point of view of the microcode. Although, this approach has many advantages, it has one big disadvantage: it is very easy to make mistakes, it is very easy to forget one of the parallel paths of the step.

In order to look at this procedure, let us analyze one instruction. We will start with a simple instruction like MOVE (R),R. That is, the content of the first register is used as an address and the content of that address in memory is loaded into the second register.

In the execution of this instruction, after the FETCH we have the following steps:

The arithmetic logic unit passes the content of register A with the arithmetic logic unit driving the data bus and the memory address register latching what is on the bus; the Am2910 has CONT as its instruction;

The next step is driving the address bus with the memory address register, the data bus is driven by memory, the arithmetic logic unit has an instruction D OR zero and the result is latched in register B. We are assuming a single memory cycle; the Am2910 has a JMZ, jump to zero, to fetch the next instruction.

The purpose is now to compute the time it takes each one of these steps and, after doing the same with all instructions, we select the clock.

The first step requires for the address A in the instruction register to propagate and the content of A to pass to the memory address register. This takes the delay from A to Y in the Am2901 and the settling time in the memory address register. Looking at the table of combination propagation delays we find the time delays for A bypass to Y, instruction to Y, clock to Y, etc.

Notice that in this instruction we have 40 ns for the time for passing A and it looks like we could use a short clock. We can see now that if the clock is 40 ns the Am2910 will not have time to do anything because we know that even the continue takes 98 ns. So, there is no time for the Am2910 to work and the time for the cycle should be 98 ns and not 40 ns as a superficial analysis will produce. This is a very important point to consider for each of the step in our instructions.

The fundamental point is to find where the cycle starts. One point to start is in the instruction register for the address A. This point shows us that we have to be careful how do we do these things. The question here is if we have to start the moment the instruction is latched in the instruction register or if the instruction register latched in a previous clock and we do not need to consider the propagation delay inside the instruction register.

Let us look further into this problem to develop a clear idea of what is happening in our computer. We

need then to look at the fetch. The fetch starts by the address bus driven by the program counter, the data bus driven by the memory, the instruction register latches at the end of the cycle and the program counter increments after the end of the cycle.

So memory is being accessed and at the end of the clock the data is latched in the instruction register. Then what? What happens next? The Am2910 has a signal that opens the MAP ROM, the data in this ROM propagates through the Am2910, from there propagates through the microcode ROM and at the end of the clock the pipeline register latches this data. The important question we have at this moment is, what are we latching in the pipeline register? It is clear that the pipeline register has the first step of the instruction to be executed. The question that remains is which instruction is the one that will be executed? If you analyze carefully what is happening, you will find that the instruction that has been loaded in the pipeline register is not the one that was just fetched but the instruction the was sitting there from the previous fetch.

This is a very important point. Note that simultaneously, in parallel the address in the program counter propagates through memory and the data in memory propagates and at the end is latched in the instruction register. At the same time, the instruction sitting in the instruction register has propagated through the MAP ROM and the content of this ROM propagates through the Am2910, the microcode ROM, and at the end of the clock the content of this ROM is latched in the pipeline register.

The efficiency of the code depends on how good we are in making them work at the same time. Note then that we have a pipelining in the instruction register that we have not considered before and that we were very proud of not having. We have to consider it now, especially when we have a conditional instruction that fails, the common problem to all pipelining. Notice that this affects which registers are actually used in the execution of every instruction.e to add one step to the fetch and change the Am2910 instruction from MAP into CONT, and say MAP in the next step. Interesting things to consider are first, how easy it is to get tricked into considering that the computer is doing something that it is really not doing; the second problem is when you study how long the computer takes to execute an instruction, it is very easy to forget that several things are happening in parallel and consider as the time delay for a step only the time delay in one of the paths when some other delay might be longer.

Now, we have another point. Recall that our computer is using part of the instruction register for the address of the register to be used and for other purposes. Since when we start execution of an instruction we latch the next instruction, it is necessary to have another register for the instruction being executed. Now we see that the double register is mandatory if we want to use this pipeline.

One interesting point to make again is that these details do not appear in the design until you do the timing analysis of the computer in detail. There is no way for this instruction pipelining to appear before and it has been necessary to come to the timing analysis to find it. Note also that this problem was not apparent when we made the timing analysis in the circuit but it became apparent when we analyzed the timing of the instruction steps. This is the value of the timing analysis of the instructions step by step.

Going back to the analysis of the instruction, there is something else that is interesting: consider that the

propagation from the A register to the memory address register takes only 40 ns and the cycle is 98 ns. The question that arises is, can we use the other 58 ns to start propagating the address of the memory and thus gain time in the memory access? Note that now we are talking about something else than having two things working in parallel, or pipelining; what we are studying now is the possibility of overlapping cycles. Since the first cycle has not ended, the question is if we could start the next cycle before the previous one has ended. This only requires that in the first cycle we select the memory address register as the device that drives the bus if we find a way of latching the memory address register as soon as it is possible, or we can use a transparent register. This requirement is quite easy to satisfy. Note also that if at the moment of the clock, the microcode has the same value for a given bit as in the previous clock, there is no glitch or transition at the moment of the clock. Simply, the pipeline register latches the same value it has in its outputs and noting changes.

Note that this arrangement has something interesting. Recall that the next cycle is controlled by the delay in the memory, that is longer than the delay in the Am2910. Consequently, if we have a way to do this trick we will be able to use a shorter time for the second cycle, as long as it is not less than 98 ns. Note that by doing these tricks we can end up with this computer running at 100 or 110 ns. clock period.

This is one advantage we have in the analysis of the timing on the bases of the instructions, that we can do tricks in the instruction by instruction basis, not on the basis of the circuit. Note that when working with the circuit there is no way to do these tricks. Something else, these tricks depend only on the designer and not on the user, so they are totally previsible during the development of the microcode.

Going back to the instruction, in the second step the Am2910 requires 98 ns as for all steps. The memory we said takes 150 ns, the multiplexer takes 12 s, the arithmetic logic unit takes 38 ns; so the total time is 35 + 5 + 12 + 150 + 38 = 240 ns for the two cycles. So, we could give 120 ns for each cycle if this proves to be true for all instructions. From the other point of view of analysis, we will need 98 ns for the first cycle and 200 ns for the second and this will force us to use a 200 ns clock. The proper way to work without forgetting anything, is to go through the microcode and work bit by bit, so we do not leave anything out.

Now about the condition of having a clock at the right moment in the cycle to latch the memory address register at the proper time. Recall that the Am2925 produces different wave forms for the clock and we choose for the memory address register latch that one that has a transition at the moment we need it.

Notice that while the second cycle is being executed, the JMZ is being executed and the next fetch instruction is coming down the microcode ROM to be latched in the pipeline register at the end of the clock.

One important point is that the analysis of the microcode is one method for the analysis of timing in a computer. It is not the perfect method because in engineering there is no perfect method. What we have to consider is that we have two methods for this type of analysis and we can use one or the other depending on the circumstances.

Note that going through the loops we have a pragmatic approach. The clock is sized for the longest loop and the computer will work. With the analysis of the microinstructions we can get the last nanosecond out of the computer. We see that by the loop analysis we need 200 ns clock and by step analysis we could run at 120 ns clock with a little change in the circuit to clock the memory address register at the right time.

One interesting point is the operation of the arithmetic logic unit. When we consider the delay in the arithmetic logic unit we have not considered the time it takes to perform its operation. Did we forget this delay? The instruction is OR and we have not considered its delay. Recall that we are charging 38 ns for the arithmetic logic unit because the table gives us this value for the propagation of the signals. These 38 ns include performing the operation to produce the result, making it ready to storage at the clock. Note that we start counting these 38 ns form the moment we know the data is ready at the input pins of the arithmetic logic unit. So, we are considering the delay in the arithmetic logic unit to perform the operation because it is included in the values we studied.

Now we will look into a more complex operation. Let us change the instruction in the arithmetic logic unit from OR to ADD. So, the first two steps are the same as the previous one but we put the data that comes from memory into the register Q and then we will add Q + B and store to B. Now, the question is how long it takes this step we are adding. In this case we will have to consider the propagation of the carry, eight times through the eight arithmetic logic unit chips we are using. Since each step is 40 ns, it is necessary to allow 8 x 40 or 320 ns. In the add instruction we have 8 chips that from the beginning are adding Q to B plus the carry. Q and B do not change but the carry might change; so, 40 ns later the carry changes and the second arithmetic logic unit starts adding again and propagating the new result. If the conditions are such that the carry changes at every 4 bits, it will take 320 ns before the result is the real result that we want. This is what is called ripple carry because the carry ripples from arithmetic logic unit to arithmetic logic unit.

Now this will force us to use 320 ns for every time we execute any instruction where the carry could change. This does not seem right and there must be some solution to this problem. The solution exists and it is called the carry look ahead. The Am2902 is a very inexpensive chip that takes all the carry generate and carry propagates from all the chips and generates the carry in for all the chips. The only problem is that the 2902 handles 4 chips and we need 3 of them to develop a 32 bit word. The typical delay of the 2902 is 14 ns which is the maximum delay we could have, so we have to allow for 37 ns for the first arithmetic logic unit to produce the G and P outputs, 14 ns for the 2904, 14 ns for the second one, and 40 ns for the final add. By using the 2904 we need only 95 ns. So, it is important to use the carry look ahead. The larger the word the more important this is.

Although this procedure is complicated, it is not impossible to learn it and do it. It is clear that those of you that have been participating in the class should be able to attack a design of this type and take it to completion.

We have been talking about indirect addressing, direct addressing, the immediate instruction where the memory is driven by the program counter, which is ready. It should be clear that the memory access in this case takes 200 ns. Since the difference between 120 ns and 200 ns is too much, we can think in two paths,

having different clocks one at 120 ns and another at 200 ns or simply break the access in two pieces each of 120 ns. We talked about this when we studied the pre-fetch. Now, we have to think about the FETCH, we have considered it as one cycle but now it is necessary to look again to how long it will take. It is easy to get that the fetch requires also 200 ns. So the Am2910 puts a limit from below in 98 ns or 100 ns as the minimum; memory is putting a limit from above at 200 ns minimum. The fetch is another instruction that can be very easily broken into two cycles, or have two clocks selected by the microcode.

Now, let us go to another extreme, the immediate short, and we have to be careful. In this instruction we drive the data bus with the low part of the instruction register and we latch in the arithmetic logic unit. So the delay in the arithmetic logic unit is 38 ns. we have 13 ns for the multiplexer or buffer, so we have 51 ns for the arithmetic logic unit side of the instruction and since we have 98 ns in the Am2910 side of the instruction we need to use the 98 ns. So again we find the Am2910 limiting the time.

Note that when considering the multiplexer there is the possibility that in a given cycle the select lines have been set before the start of the cycle and the data is the one that will be setting; consequently, the time we should consider is the one from the D input to the Y output, or from the select to the Y depending of which one is pertinent to the case.

8.5.- **Analysis of Pipelining, and Cache**.

There is one point from the analyses we have made that deserves mention. In the current literature you have read there is a large concern with the problem of what to do when a branch instruction appears, what they call the "bubble" of instructions that must be destroyed to process the branch instruction. It should be clear that all the authors of computer designs put a large emphasis in developing very complex schemes to reduce the impact of the "bubble" in their computers.

The point to ponder is that in our design we have not considered this problem. The question is, why we don't have that problem in our design. How come other designers have to go through elaborate procedures to solve the problem that we have not even mentioned. Are we doing something wrong? are we overlooking a problem we should be considering? It is evident they are pipelining and we are not. We talk about pipelining exactly in the place of the problem which is fetching an instruction ahead of time, vs. overlapping the fetch.

This is a philosophical point. When you do something one way and everybody else does it differently, it is logical to think that you are doing something wrong, that you are overlooking something and it is important to check. If you find the reason why everybody else does it differently and you are right, then you can proceed with confidence.

At this moment we have our design complete, with the exception of the timing; in such a way that we can build it. The situation is that everybody else is still considering very complex architecture. We do not consider the use of cache memory simply because we don't need it. Where can we use cache memory? there is no place. Notice that in our computer, the way we are considering it today, the only advantage we can have by using cache memory is that the fetch will take one cycle instead of two. This is true only in

the fetch of the next instruction and in the very few cases where we have two word instructions. In these cases we will gain one cycle per memory access. We have considered the possibility of having two instruction registers. This will permit us to pre-fetch many of the instructions, eliminating this advantage.

Cache memory is a name used for a block of very fast memory between the main memory of the computer and the computer itself. You understand there is a controller that is guessing what the computer will need and puts it in the cache. The big problem: the cache must be filled at the speed of the slow memory. The larger the cache, the longer it takes to move it. So, if at a given moment the computer finds that the cache memory does not contain the piece of data that is required, it is necessary to wait until the controller refills the cache memory. In more advanced controllers, it is possible to get the controller to find the right piece of data, or to let the computer bypass the controller and find it itself. All this requires arbitration, meaning time.

The use of cache memory is then debatable when the advantage of using the cache memory decreases very much when the size of the cache increases or in the cases where the computer changes path very often. The other problem is that every time the computer goes for memory, there has to be hardware that compares the addresses of memory existing in the cache with the one required by the computer, and this extends the length of time of the memory cycle. Another problem is memory modification. Before destroying the contents of the cache, they have to be saved if there was any modification; consequently, the controller has to keep track of the modifications. The fine point is that as the overhead increases the advantages of the cache memory decrease. The other aspect is that the advances in technology are producing faster and faster memories. Today there are inexpensive static memories that have cycles on the order of 100 ns. When these memories are used for main memory, it is very difficult to justify the use of cache or any of these arrangements.

A different story is what we talked about with the double instruction register. When we have an opportunity to pre-fetch the next instruction we do it. We need to recognize that this will happen only in very few cases because all our instructions are very short, because we are adopting the RISC criteria with very good results. In the case of a jump or branch instruction, we simply ignore the pre-fetched instruction and we include the fetch of the next instruction in the code of the jump or branch. The instructions that permit the pre-fetch transfer the content of the auxiliary register into the instruction register at the proper time, and they end up in jump map, instead of jump zero. The instructions that do not permit the pre-fetch end with the JUMP ZERO and perform the normal fetch. This is a more proficient arrangement that the use of cache memory.

So, this is the reason that we don't have the problem of the "bubble". This is because we are considering we will be using a fast memory as it is the normal memory of today technology. This point is that an engineer needs to continue advancing, continuously learning to maintain on top of technology, to abandon criteria that were good when you were studying but that the advances of technology have made obsolete.

THE DESIGN OF COMPUTERS USING BIT-SLICE TECHNOLOGY

Michael E. Valdez, Ph.D., P.E.

SLICE #9

Slices: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9

SOME ADVANCED IDEAS

This Slice is about certain limitations that are included in every computer simply because of the sheep philosophy that forces everybody to follow what everybody else is doing. We want to talk about a different approach to computers because you will be living in the future. You will not be living in the past or in the present, you will always be living in the future. So it is good for you to think a little bit in a different way.

9.1.- **An Advanced Idea**.

What we want to point out is certain limitations that are included in every computer because of inertia. When you stop and think, you find there is no reason why they are included except for the fact that everybody does it.

Let us talk about for example the circuit that we used in our design where we have done the same thing because we wanted to leave this idea for the end. We have the sequencer, micro code, MAP, and instruction register, etc. We also have the loop of the instruction that is what makes the computer work. The input of the instruction register is connected to the data bus so the normal operation of the computer is to develop the content of memory from the memory address register and store it in the instruction register.

Now, the point we want to make is what we call instruction. The fundamental idea of today is that the instructions have to be very elementary, what we call the assembler language of the computer and, almost by definition these instruction have to be elementary. The question for you is why? Why do we have to have a

set of elementary instructions? Why we have to have instructions of the type MOVE this data from this register to that? etc. The answer is that it is mainly because of the inertia of the designers. There is absolutely no reason why we cannot have an instruction that performs the DO loop of FORTRAN. Actually, the author has implemented in micro code instructions of this type. With this type of instruction we do at the micro code level operation that are normally considered to be high level language. Notice that we are not talking about putting the high level language in micro code but implementing high level commands in micro code. This is currently done in controllers, like the video controller that receives a command to draw a circle; why not in computers?

9.2.- **The Stack Computer**.

Imagine we have a control unit, an arithmetic logic unit, a program counter connected to the address bus of the program memory, the normal stack and stack pointer for storage of the program counter during the subroutines, and imagine that we have another block of memory with its special stack pointer. This is what is called the Stack Computer. Please see Fig. 9.1.

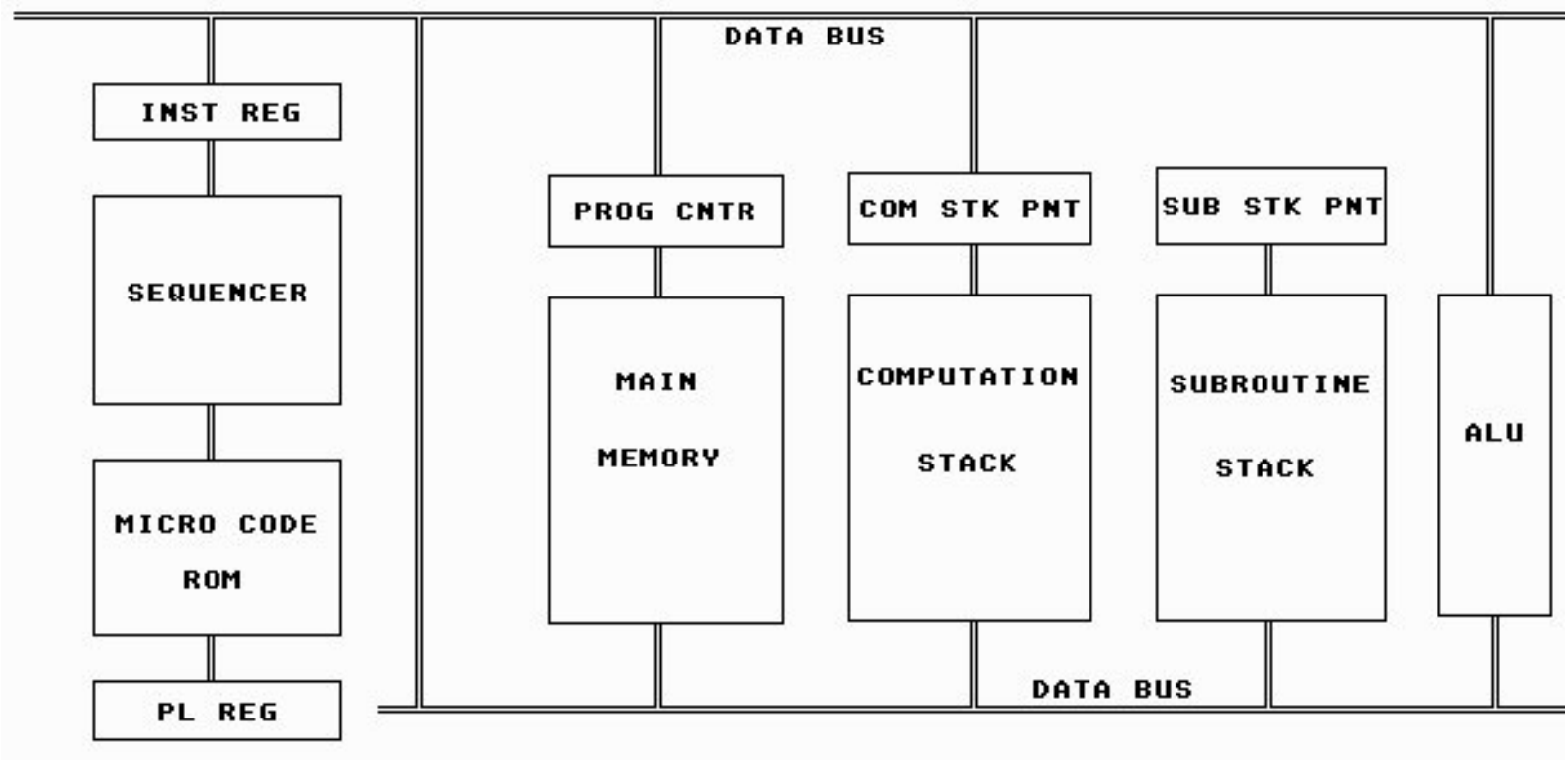


Fig 9-1

The stack computer has several important differences with the normal computer we are used to. One big advantage is that as soon as you consider the stack computer you do not have an address bus. What is the address bus in the normal computer becomes simply the connection between the program counter and memory.

Another concept that disappears in the stack computer is the concept of operands. It is not necessary to specify where the operands are or where they should be. The only thing to be specified is the operation to be performed. For example, you have an instruction ADD. You may question add what? The answer is always

the top elements of the stack, since add requires two operands, the top two elements in the stack are added. The result of the operation is always stored on the top of the stack, after destroying the operands.

The instructions are very simple, the computer has the same capabilities of any other computer, and the operation of such a language is extraordinarily fast because there is very little overhead in the operation of the computer. Note that since there is no operand addresses, there is no time spent in getting the address from wherever it is.

If the stack pointer covers the whole memory; that is if the stack pointer is a 12 bit counter and we have 4 K of memory, or if the stack pointer is a 16 bit counter and we have 64 K of memory, you do not care where your stack starts and you could work without ever initializing the stack pointer. So we need only increment and decrement the stack pointer. If the stack pointer is not commensurable with the memory, then it should be initialized and its value tested to prevent it from going out of the available memory. Note that this stack is not the one used for storing the return address of the subroutines, but the computation stack. There are several stacks in this computer, minimum two, one for the returns and another for the computations.

In this type of computer we normally do not need a memory address register because the main memory is normally only used for the program. When the application requires storage memory then it might be good to have another section of memory devoted to random storage and that one will have its own memory address register.

There is a language, FORTH, that simulates the stack computer in a normal computer. FORTH is a language that was developed by Charles Moore, a computer programmer who got tired of the redundancies, and the limitations of the computer languages. He was working at the time in the National Radio Astronomical Observatory, so the language has many interesting characteristics, one of the most important ones is that it is the fastest high level language. The language simulating a stack computer in software runs almost at the speed of the assembler language for the computer in which it is implemented. If you compare it with other high level languages, the difference is enormous. Charles Moore, in cooperation with others, has now developed a computer that works in FORTH, the Novix NC4000.

9.3.- **Stack Computer Instructions**.

Now, how do we implement a high level instruction in micro code. First, we could have an assembler, as it is normal with all programs, that reads the input and converts the commands into codes equivalent to the instructions or opcodes we have been talking about. We could also go the iterative route where the micro code has commands to read the keyboard and interprets the input stream as commands.

Let us start by analyzing some instructions for our computer and, doing the same as we have done before, having a coding form similar to the one we used before, with address driver, data bus driver, computation stack pointer, computation stack read/write, return stack pointer, return stack read/write, program counter, memory address register, memory read/write, who latches the data, arithmetic logic unit instruction, Am2910 instruction, etc. So consider we want to implement the instruction ADD; that is, add the top two numbers in the stack.

First the data bus is driven by the computation stack; the computation stack pointer is incremented at the end of the clock, the arithmetic logic unit will latch and the arithmetic logic unit has the instruction D pass Q, the Am2910 has continue.

The second step we leave the computation stack driving the bus, we will not change the stack pointer in this step because we need to store the result, the arithmetic logic unit latches, the arithmetic logic unit has the instruction D+Q to Q because the data bus is busy.

Finally we change the computational stack to write, the pointer is not changed, the stack latches, the data bus is driven by the arithmetic logic unit.

Note that there are no registers and the register addresses are connected to the micro code and are used only by the micro code. Note also that the instruction we have developed is comparable to the add memory to memory with the result stored in memory, an instruction that is normally quite complex and long because we need to read the address of both operands. This instruction is almost as fast as the register to register in the computers we have been studying.

Let us look now into another instruction that is very different from the instructions that are normally implemented in a computer, this is the instruction SWAP, which is a typical stack instruction. The instruction SWAP inverts the order of the top two elements in the stack. This instruction is very useful in subtraction to change the order of the two numbers we wish to subtract.

The first line is the same as before. The second line is similar in which we move the second element in the stack to register 0 in the arithmetic logic unit and the Am2910 continue.

The next steps moves the content of register Q back into the stack so the arithmetic logic unit drives the data bus, the stack writes, the stack pointer is decremented, the arithmetic logic unit has the instruction Q OR 0, and the Am2910 has continue.

The final step moves the content of register 0 back to the stack. For this purpose we need to maintain the arithmetic logic unit driving the bus, the stack will still write, the stack pointer is not changed, and the Am2910 has zero. Note the use of all the registers in the computer as auxiliary registers for the micro code.

There are other instructions typical to stack computers like ROL that changes the position of the top three elements of the stack. There are two ways for implementing this instruction: one is to save the top of the stack in the arithmetic logic unit and then move the next two elements to their new positions and finally bring the element that was stored in the stack to its own position. This procedure has the problem that a lot of time is spent changing the address of the stack. Another way to implement this instruction is to bring the three elements to the arithmetic logic unit and then put them back in the stack in the new order. Note that the execution of this complex instruction takes only 6 microcycles plus the fetch. The other procedure takes 9 microcycles. Performing this operation using a normal assembler language will take at least six instructions, each one taking several cycles.

Another possibility is the PICK that permits to get a copy of any value in the stack and put it on top of the

stack. To execute the instruction we need to increment the stack pointer n times, pick the value, then decrement the pointer n+1 times, or save the stack pointer, and put the value back as the new top of the stack. Note that we could use the arithmetic logic unit to add n to the stack pointer and later subtract n+1 from the stack pointer, but this requires paths from the input and output of the stack pointer to the data bus, that up to this moment have not been considered. Since this is one of the few instructions requiring these paths, they are not used.

Now we wish to look into the DO loop. The DO loop is not trivial. We are talking about the typical FORTRAN DO loop. We can not use the counter of the Am2910 because in any DO loop we want to have access to the counter of the loop. We could say for example, print I, or I+A, or any other use of the index of the loop. In FORTH the problem is solved by calling I the top of the return stack; that is, there is an instruction I that copies the top of the return stack into the computational stack that is very easy to implement in micro code. Then, when the loop starts, we put the pointers for the loop in the return stack. There is naturally the restriction that we cannot have incomplete operations inside a DO loop.

The statement in FORTH is 1 7 DO xxxxx LOOP. The computer finds the 1 and puts it in the stack. Then it finds the 7 and puts it on top of the stack. Then it finds the DO and it has to enter into the loop so let us bypass that operation for the moment. Then it performs the operations of the loop and finally it reaches the instruction LOOP or CONTINUE.

At this moment, the computer has to increment the index, compare it with the end, if I is equal to or larger that the end of loop, the loop has been satisfied; if it is not equal, it goes back to the beginning of the loop. Then when executing the DO statement we need to save some place, in a loop stack for example, the address of the beginning of the loop. Inside the DO loop we have the right of changing the computation stack, so, we have to move the two pointers to another stack. Note that all we have said can be done in micro code. The address of the beginning of the loop is only the program counter at that moment, so we need a path from the output of the program counter into the data bus which is also used by the calls for subroutines. The program counter must also be loadable, so we can produce jumps.

The DO loop is very similar to the call for subroutine in the Am2910 that does not destroy the return address. It is very easy to perform the loop now, we increment the top of the stack, change the pointer, move the value to the arithmetic logic unit, compare them and at the same time move the pointer again. If the loop has not ended, the loop stack is copied into the program counter; if the loop has ended then the pointer to the loop and return stacks are moved up destroying the pointers and the return address and the program continues with the next address.

This is one of the most complex instructions for a stack computer as you can easily see. You see that everything can be done in micro code.

Note that if we have an instruction to copy the top of the return stack to the top of the computation stack, we can obtain the value of the index for whatever use we have.

So we see it is possible to develop a stack computer whose primitive language has high level instructions.

9.4.- **High Level Microcoded Instructions**.

Now, imagine that we do not have a stack computer but a normal architecture and we want to see if we can implement the DO loop in micro code. We can as long as the hardware has access to some of the registers where the programmer cannot access. For example, we can take two registers to store the index and the end of the loop and another for the return address. A big restriction arises that we cannot nest do loops. In the stack computer it is possible to nest loops. Another solution is to use two registers as stack pointers and assign an area of memory to the micro code and proceed the same way as with the stack computer.

Let us implement the DO loop in this form in the computer we were studying before. The data bus is driven by the arithmetic logic unit, the address bus with the stack pointer, and we write in memory. We are storing the registers in the stack and the stack pointer will decrement at the end of the clock. There are two ways to work with the stack pointer, one where it points to the first empty location and the pointer is decremented after storing and incremented before reading. The other operation is the opposite where we decrement the stack pointer before writing and increment after reading. This method has the advantage that the top of the stack can be read without changing the pointer.

In the same way we store the other two registers used by the DO loop, in the stack.

Then, we want to store the two indices for the loop. These indices must be in the program memory just after the instruction DO and consequently we drive memory with the program counter to get them to the two registers. Next, we proceed in the same way to move the program counter into another one of the registers.

Now we are ready to continue with the operation of the program.

In the execution of continue we increment the register that contains the index and compare to the register with the end of the loop. If the loop has ended, we pull the three values from the stack to the registers, and continue. If the loop has not ended we move, without destroying it, the content of the proper register to the program counter; that is to say, we execute a Jump indirect through that register.

So, the implementation of complex high level instructions is perfectly possible in any type of computer.

In the same way it is possible to work with the IF --- THEN ---- ELSE ---- ENDIF construct. The computer executes it by ignoring the IF but only marking that an IF is being executed, if necessary for testing purposes. When the THEN is found it is interpreted as a branch on condition. If the condition is not zero, the computer continues, if not it searches for the ELSE. If the computer executes the THEN it works normal until it finds the ELSE, which is then interpreted as an unconditional branch to the ENDIF. In any case, when the computer finds the ENDIF it erases the pointer that indicates it is working in an IF construct. That is, when the computer finds the ENDIF, it continues normal execution. The two branches can be done simply by a micro code that advances the program counter and reads the value until it finds the given opcode.

The above conditions refers to the case where the computer works without a compiler, only an interpreter that looks at the input string. If the computer works with a compiler then the compiler changes the construct into the proper branches and the micro code becomes very simple. This construct is then changed into -------

Branch if true to ELSE --------- Branch Unconditionally to ENDIF ELSE ------- ENDIF.

So, we have several examples that building a computer that works with high level instructions is not only possible but desirable. That it is not necessary to develop a computer always working with a low level, primitive, simple language. The advantage naturally, is speed.

Finally, it is perfectly possible to have a compiler in micro code that executes the program in two passes, first it compiles it and stores it in some place and then executes that code. Something like the normal operation of a FORTRAN program but all executed at the micro code level. It is possible that the user might save the compiled program for later use, as it is possible in the software version.

The micro code is a very powerful tool for the design of computers, because it has many of the characteristics of hardware and of software. It is really hardware but it has most of the advantages and few of the limitations of software.

We have now the problem of the word size. When we talk in assembler language with the complex instructions that perform simple operations, there is a big advantage in a long word so the complex instruction uses most of the time one word. When we talk about high level language we have the problem that we have simple instructions that perform complex operations. Take the case of the DO, that is very simple and performs quite a complex operation. In this case we might want to have a short word, 8 or at the most 12 bits; probably 8 will be enough. So, in this case, there is another different idea, the idea of separating the program and the data with different memory words for data memory, and for program memory. So we can end up with 8 bits for program memory, 64 bits for data memory, and 24 bits for address.

9.5.- **Some Final Thoughts**.

The picture for a person who wants to design with bit slice is bright, because we are not limited to AMD as the only source. All these chips have second sources which assure the supply. The problem is that for all practical purposes we are limited to the ingenuity of AMD engineers. There are very few other manufactures designing bit slice chips. Intel manufactures a couple of chips, Motorola is the most aggressive at the moment, increasing their number of chips for bit slice, Texas Instruments produces a few chips, Analitic Memories, Signetics and Fairchild also produce some chips. Nobody produces the whole line of devices like AMD.

Something else that is interesting to consider on the topic of fast chips is, for example multipliers. As long as you satisfy the conditions of the chips, there is no reason why they cannot be used in a non-bit slice computer. Although you will not be able to work as fast as the chip works, it will be much faster than any other solution. You will naturally have to fake the micro code.

Something else that is interesting to mention is that we have been talking about computers and controllers but there is absolutely no reason why we cannot use bit slice to design digital filters for example, an on-line digital filter. A digital filter involves multiplications and additions and a few locations of memory. So, when the situation justifies it, it is possible to design digital filters that work very fast, as when the sampling rate is high. The storage needed will depend on the length of the filter.

In the same way we could develop, for example, a co-processor for linear predictive coding, or whatever we want. A chip exists for obtaining the linear predictive coding of a continuous sample of speech, but if we want to develop linear predictive coding for a higher sampling rate, bit slice could provide a solution to the problem. Many other applications can be found as co-processors, pre-processors, post-processors, etc. The area of application in controllers and computers is naturally the area for which these devices have been designed for.

The question that remains is if bit slice is the only alternative that we have for the design of large computers. Note that the solution with multiprocessing has the problem which is similar to an office with one girl working there and you get some work done. If you add another girl you do not get double of the output, you get much less; another girl added does not increase the output in proportion to the number of girls but each one that is added, increases less the output. A moment gets where the girls start running into each other and the output goes to zero. The same thing happens with multiprocessing.

The pressure for bigger and larger computer is there. What we have developed is not top of the line; 100 ns per instruction is not too fast. The question is what avenues you have for developing a top computer. The only other avenue is to design the computer and building the pieces in large scale integration so they work as fast as possible.

You have to consider that the normal devices are getting faster; consider the 74Hxx family is CMOS devices with high speed and low power consumption. Recall that the bit slice has a large power consumption, each chip Am2901 takes 265 milliamps. Our computer has eight of them and they alone take 2 amps. This is a lot of heat to take out of the case.

I imagine that now that CMOS is getting faster somebody will come with a family of bit slice in CMOS.

9.6.- **The Internal Buses**.

One problem we have not talked too much about is the problem of the buses. We have only one bus in our control unit. This is not necessary. It is possible to have several buses and have connections between the buses so we can parallel certain operations when the buses have been properly divided. This requires an analysis of what operations can be performed in parallel if the buses are divided. The reason we did not analyze this problem is that this does not correspond to an introductory course. The other extreme to our computer is not to have buses but a switching network in which every device can be connected to every other device, without affecting the operation of the others. A large switching network can become so complex that the added delay in the multiplexers reduces the efficiency of the unit.

9.7.- **A Summary**.

It is interesting to see what important concepts we have studied:

- Pipelining and all related concepts;
- Microcode and related topics;

- General architecture of computers;
- The execution of the steps;
- Synchronous and asynchronous operation;
- Design with a computer and designing a computer;
- Design of the control unit;
- Ways to implement the control unit;
- The micro coded solution;
- The hardwired solution;
- The concept of bit-slice;
- The sequencer;
- The arithmetic logic unit;
- Architectural details;
- The Program Controller;
- Location of the program counter;
- The problem of the buses;
- Location of the stack pointer;
- Location of the memory address register;
- Use of intermediate registers;
- The problem of timing;
- Timing of the operation of the steps;
- Timing by delay through all paths;
- Analysis of the bottle necks;
- Microprogramming;
- The instruction set.