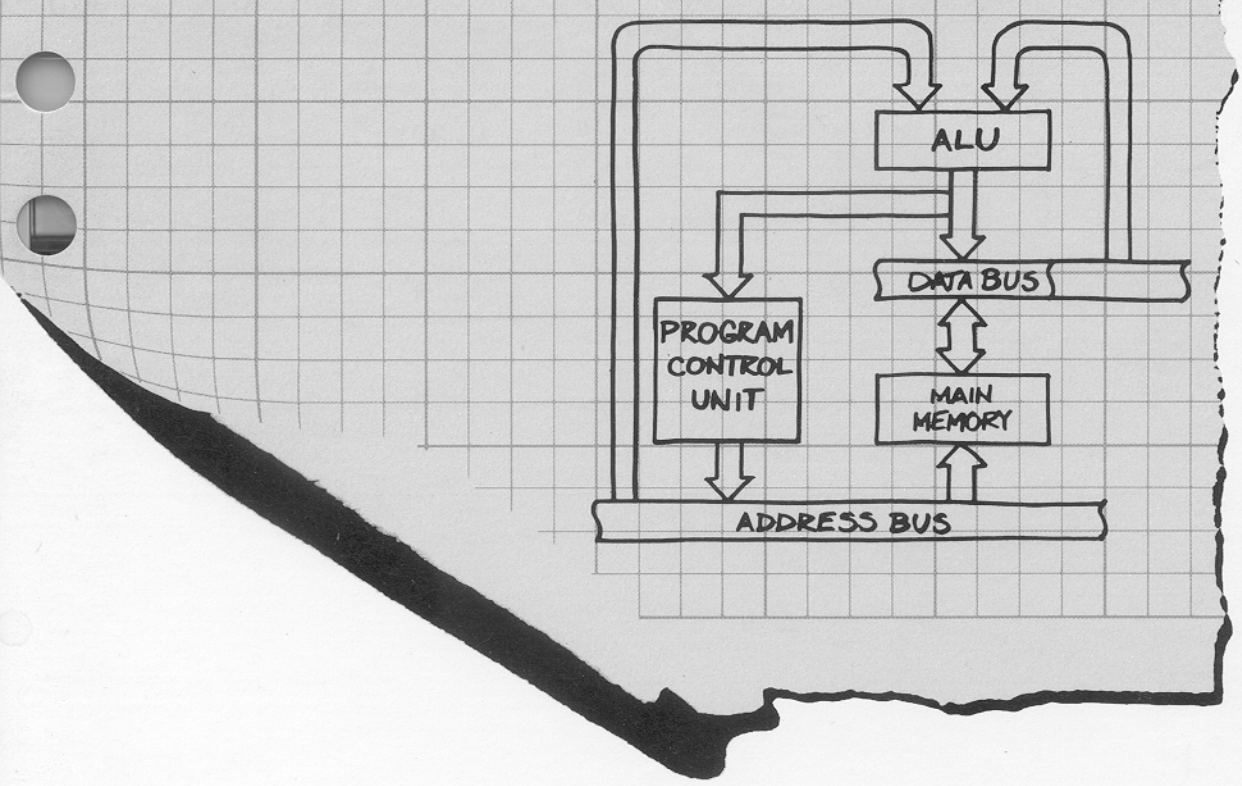

Build A Microcomputer

Chapter V
Program Control Unit

Advanced
Micro Devices

Advanced Micro Devices

Build A Microcomputer

Chapter V
Program Control Unit

AM-PUB073-5

Introduction

In order to access instructions and data in an orderly manner within a computer, a Program Control Unit is usually used to provide the most efficient mechanism for program control. A program is a set of instructions which direct the processor to perform a specific task. Ordinarily, program instructions are stored in sequential memory locations. During the normal processing of a program, an instruction is fetched from the location specified by the program counter, the instruction is executed, the program counter is incremented, and another fetch and execute cycle begins. The addressing mechanisms that such control unit might employ are various. Indeed there are some machines that literally use dozens of addressing modes to fetch instructions and data. In this discussion of program control units, several of the addressing modes and their common implementation techniques will be discussed. The addressing modes used commonly in today's machines include register, immediate, direct, indirect, index, and relative and various combinations thereof.

Data Formats

Technically, an instruction set manipulates data of various length words. Generally speaking, most 16 bit minicomputers can manipulate data of three different word lengths: 8-bit bytes, 16-bit words and 32-bit double words. This data may represent fixed point numbers, floating point numbers, or logical data. The data is used as operands for the instructions, and is manipulated as indicated by the particular instruction being executed.

Typically, fixed point data is treated as signed 15-bit integers in the 16-bit representation or as signed 31-bit integers in the 32-bit double length notation. Positive and negative numbers are represented in the ordinary 2's complement notation with the sign bit carrying negative weight. Positive numbers have a sign bit of zero and negative numbers have a sign of one. The numerical value of zero is always represented with all bits LOW.

Floating point numbers consist of a signed exponent and a signed fraction. Many different formats are used by manufacturers in expressing floating point data and these variations will not be described here. Let it simply suffice to say that the floating point number represents a quantity expressed as the product of a fraction times the number 2 raised to the power of the exponent. In some cases, the number 16 is raised to the power of the exponent. Typically, all floating point numbers are assumed to be normalized prior to their use as operands. No pre-normalization is performed and all results are post-normalized. Usually, the floating point instruction set will normalize un-normalized floating point numbers.

Logical operations are used to manipulate 8-bit bytes, 16-bit words or 32-bit double words. All bits participate in the logical operations.

Instruction Formats

Various minicomputers use different types of instruction formats ranging from the very simple straight forward formats to the more complicated difficult to decode formats. For example, a register to register format can consist of a simple 8-bit opcode and two 4-bit source operand specifiers. On the other hand, it may consist of a byte or word specifier, an opcode specifier, source and destination register specifiers, and mode specifiers for each of the source and destination register selections. Again, it is not the purpose of this application note to describe all of the trade-offs in selecting instruction formats but rather to select a simple format such that the student of bipolar microprogrammed microprocessors can understand the techniques used by instructions for operating the machine.

Thus, we will use a few 16-bit and 32-bit formats in this application note to demonstrate the function of the program control unit in various types of instruction execution.

Instruction Types

For purposes of this application note, we will define nine different instruction types using various addressing modes. As we define these instruction types, we will use the basic ADD instruction as the example in all cases. It should be recognized that the operations of the instructions are similar for all the arithmetic as well as logical type operations. However, by using the ADD instruction it will be easier to describe the operation of each of these instructions rather than to try to be very general in their description. Figure 1 shows all nine instruction types with their appropriate names. As is seen, four of the instruction types are single 16-bit word instructions while five of the instruction types are double word or 32-bit, instructions. The advantage of the double word instructions is that a second word can be used as an address whereby it provides an index value or a second word can be used for data which is used as an immediate value.

Register-to-Register Instructions

When the register-to-register (RR) instruction is executed, it is simply a technique for selecting two of the machine's internal working registers in order to execute the desired operation. The instruction is fetched from memory and placed in the instruction register and the source register R2 and second source register R1 are selected as the two source operands for the ALU. Register R1 is the destination register in addition to being a source register and the results of the ALU operation will be placed in the register specified by the R1 field. In the instruction format shown in Figure 1 for the register-to-register instruction, the 8-bit opcode field specifies the machine operation to be performed. The next 4-bit field, R1, in the instruction format specifies the address of the first operand. In most machines, the R1 field is normally the address of a general register. The 4-bit R2 field in the register-to-register instruction format specifies the address of the second operand; this also is normally the address of a general register. In most machines, the R1 field also in addition to being a source operand is the destination general register select. Thus, the results of the operation are stored in the register selected by the R1 field.

The RR instructions are used for operations between registers. We are assuming in this discussion that the machine contains 16 general registers which function as accumulators or index registers in all arithmetic and logical operations. Each general register contains a 16-bit word consisting of two 8-bit bytes. For arithmetic operations, the most significant bit is considered the sign bit using 2's complement representation. The general registers of the machine are usually numbered from 0 to 15 (decimal) and written in hexadecimal notation as 0 through F. In this example, the general registers have not been given specific functional assignments. However, in some machines certain registers are assumed to perform specific functions. These can include specific stack pointer registers and program counter registers. Figure 2 depicts the typical signal path for executing the RR instruction in a bit-slice system.

The actual operation of the Register-to-Register Instruction is as follows. First, the instruction is fetched and placed in the instruction register as shown in Figure 2. This is part of the fetch routine. Next, the instruction is decoded via the mapping PROM and the appropriate microinstruction in the microprogram memory selected and placed in the pipeline register. Then, the instruction is executed where the two registers in the general purpose registers of the Am2903 are selected by the contents of the R1 and R2 fields of the instruction register. The actual microcode required to

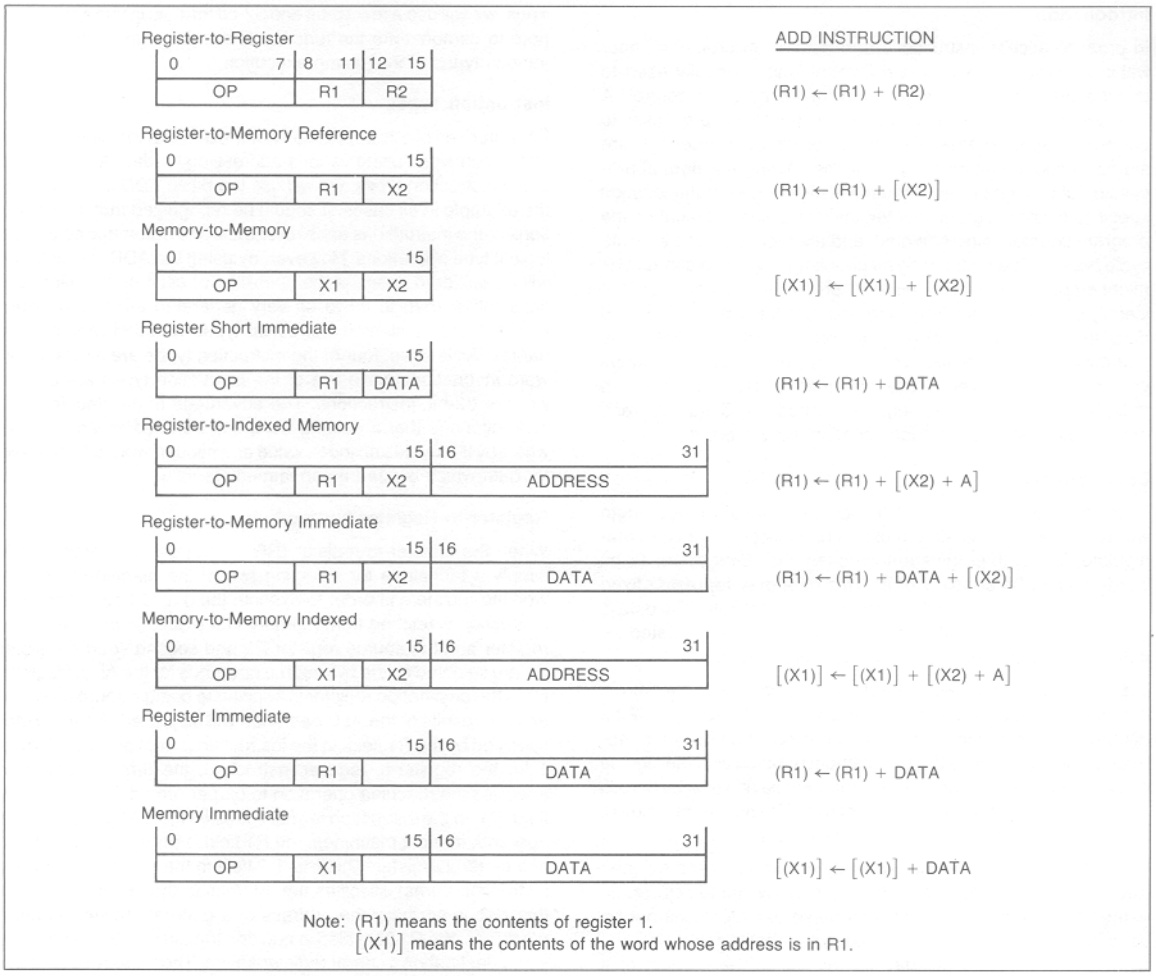

Figure 1. Various Instruction Types for the ADD operation.

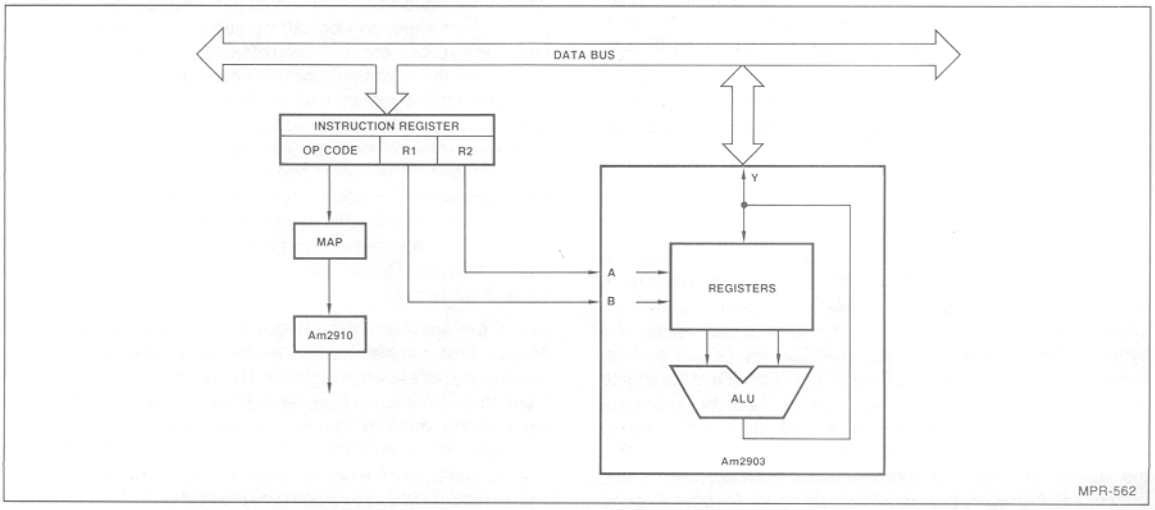

Figure 2. Register-to-Register Instructions Select Two Registers in the Am2903 Array for Instruction Execution.

execute this instruction is shown in Figure 3. Here, we assume the Program Counter (PC) value is contained in one of the general registers and can be selected by microcode as well as the R1 and R2 fields. This was shown in Chapter 3.

Register-to-Memory-Reference

The register-to-memory-reference instruction is one whereby the contents of the memory location pointed to by the register identified with the X2 value is fetched from memory and then added to the register value specified in the R1 field. The result of this operation is placed in the register specified by the R1 field.

Figure 4 shows a general block diagram of the hardware used to implement the instruction types described in the first part of this application note. As shown, the memory address register can be driven by either the Y outputs or the DB outputs of the Am2903s.

In addition, the Y outputs of the Am2903s can be placed onto the memory data bus by means of a three-state buffer. The computer control unit is intended to be representative of that described in Chapter 2 of this application note series. For purposes of this discussion, we assume the program counter (PC) is one of the general purpose registers within the Am2903 register stack. Later, we will change this concept and use the PC external to Am2903.

The operation of the register-to-memory-reference instruction as depicted in Figure 1 can best be described by referring to Figure 5. Here, we see the first three microinstructions that represent the fetch routine for the currently described machine. First, the program counter is placed in the memory address register and the program counter is incremented and returned to the PC register.

Microinstruction Operation	Microcycle Time											
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	X												
Fetch Inst to IR		X											
Decode			X										
R1+R2 → R1				X									

Figure 3. Register-to-Register Instruction Microcode.

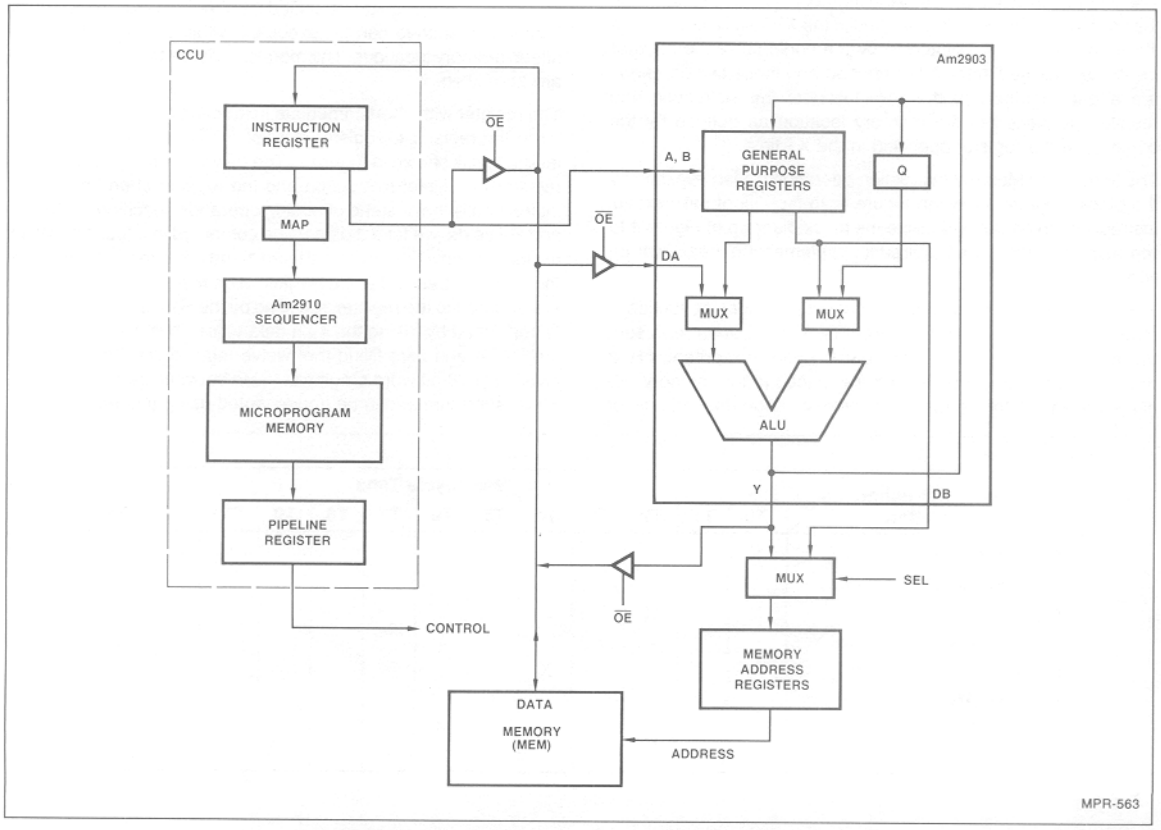

Figure 4. Simple Memory Addressing Scheme with PC in the ALU.

Microinstruction Operation	Microcycle Time												
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	X												
Fetch Inst to IR		X											
Decode			X										
(X2) → MAR				X									
MEM + R1 → R2					X								

Figure 5. Register to Memory Reference Instruction Microcode.

Next, the instruction is fetched from memory and placed in the instruction register within the CCU. Thirdly, the instruction is decoded via the mapping PROM and the appropriate microinstruction selected and placed in the pipeline register. To execute this particular register-to-memory-reference instruction, it is necessary to place the contents of the register specified by the X2 field into the memory address register. Then the contents of memory can be fetched and the operand added to the value currently contained in the register specified by the R1 field. The result of this operation is placed in the register specified by the R1 field. All totaled, the execution of this register to memory reference instruction requires five microcycles as depicted in this example.

Memory to Memory

This instruction is one whereby the memory location pointed to by the contents of the register specified in the X2 field is fetched and the memory location pointed to by the contents of the register locations specified in the X1 is fetched and these two operands are added together. At the completion of the instruction, the resultant is placed in the memory location as defined by the contents of the register specified in the X1 field.

The Memory to Memory Instruction operation is also depicted by the block diagram shown in Figure 4. In fact, all of the next six instructions to be defined utilize the block diagram of Figure 4 to represent the hardware required for implementing these instructions.

The microcode required for the memory to memory instruction is detailed in Figure 6. The first three microinstructions represent the fetch routine. In the fourth microinstruction, the contents of the register specified by the X2 field are placed in the memory address register. Then, in the fifth microinstruction the contents of this memory location is loaded into the Q register within the Am2903. This value is temporarily held for use later. In the sixth microinstruction, the contents of the register specified by the X1 field in the instruction is placed in the memory address register. On the seventh microinstruction, this operand is fetched from memory and added to the contents of the Q register with the result being placed in the Q register. In the eighth microinstruction, the current contents of the Q register is returned to the memory location. This memory location is specified by the contents of the register specified by the X1 field and is still in the memory address register. Thus, we have used the Q register as a temporary holding register for the data used in this instruction.

Register with Short-Immediate

This instruction is a technique whereby a 4-bit field is added to the contents of the register specified by the R1 field. Thus, short jumps or branches can be executed within a range of zero to fifteen memory locations. The more significant 12-bits of the word are zero filled.

The register with short immediate instruction operates very similar to the register-to-register instruction. The microcode for this instruction is shown in Figure 7. The only difference between the register-to-register instruction and the register short-immediate instruction is that instead of adding operands specified by the R1 and R2 fields, we take a data value contained in a four-bit field in the instruction as depicted in Figure 1 and add it to the contents of the register specified in the R1 field. The results of the operation are returned to the register specified by the R1 field. This addition is performed by taking the 4-bit data value shown in Figure 1 as the DATA and zero filling the twelve most significant bits. This gives us a 16-bit word ranging in value between zero and fifteen. Thus, short jumps can be implemented using this technique.

Microinstruction Operation	Microcycle Time												
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	X												
Fetch Inst to IR		X											
Decode			X										
(X2) → MAR				X									
MEM → Q					X								
(X1) → MAR						X							
MEM + Q → Q							X						
Q → MEM								X					

Figure 6. Memory to Memory Instruction Microcode.

Microinstruction Operation	Microcycle Time												
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	X												
Fetch Inst to IR		X											
Decode			X										
R1 + Data → R1				X									

Figure 7. Register Short Immediate Instruction Microcode.

Register to Indexed Memory

The 16-bit word in the register defined by X2 in the instruction is added to the address that is the second word of memory. Then, this address is used to fetch an operand from memory which is added to the contents of the register pointed to by R1. The results of this operation are then placed in R1. The instruction format for this instruction was shown in Figure 1.

The Register to Indexed Memory Instruction is shown is Figure 8 and executed in the following manner. First, the current PC value is placed in the MAR and PC + 1 is returned to the PC register. Next, the instruction at this memory location is fetched and placed in the instruction register. On the third cycle this instruction is decoded and the contents of the microprogram memory placed in the pipeline register. On the fourth microinstruction, the PC value is again placed in the MAR and PC + 1 is returned to the PC register. On the fifth microinstruction, the value at this location in memory is fetched and added to the contents of the X2 register with the result being placed in the MAR. And on the sixth microinstruction, the operand pointed to by this address is fetched and added to the contents of R1 with the result being placed in the register pointed to by the R1 field of the instruction.

Register to Memory Immediate

In the register to memory immediate instruction, the contents of the memory location pointed to by the register specified in the X2 field is fetched from the memory and the data value which is in the second word of the instruction is also fetched from memory and added to it. This result is then added to the contents of the R1 register and the final result replaces the value currently in R1.

The register to memory immediate instruction as shown in Figure 1 is implemented using the microcode shown in Figure 9. Again, the first three microinstructions are the fetch routine. The fourth microinstruction is used to take the contents of the register specified by the X2 field and place it in the memory address

Microinstruction Operation	Microcycle Time												
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	X												
Fetch Inst to IR		X											
Decode			X										
PC → MAR; PC + 1 → PC				X									
MEM + X2 → MAR					X								
MEM + R1 → R1						X							

Figure 8. Register to Indexed Memory Instruction Microcode.

Microinstruction Operation	Microcycle Time												
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	X												
Fetch Inst to IR		X											
Decode			X										
(X2) → MAR				X									
MEM + R1 → R1					X								
PC → MAR; PC + 1 → PC						X							
MEM + R1 → R1							X						

Figure 9. Register to Memory Immediate Instruction Microcode.

register. Next, the operand at this memory location is brought into the Am2903's and added to the contents of the register specified by the R1 field with the results returned to that register. The sixth microinstruction is used to set up the memory address register to fetch the second word of the instruction. The seventh microinstruction brings this data value into the Am2903 ALU via the data bus and adds this value to the contents of the register specified by the R1 field. The result of the operation is placed into the register specified by the R1 field.

Memory to Memory Indexed

The memory to memory indexed instruction is one whereby the contents of the register specified in the X2 field are added to the second word of the instruction to form a new address. This address is then used to fetch an operand which is added to the operand selected by taking the contents of the register specified in the R1 field and using that as a memory address to fetch an operand. The result of this addition is then replaced in the memory location pointed to by the contents of the register specified in the X1 field.

The memory to memory indexed instruction is probably the most complicated of the instruction formats described in the application note. In all, nine microinstructions are required for its implementation. Basically, the first three microinstructions are used to fetch the instruction from memory, place it in the instruction register, and decode the instruction for initial operation. Again, the basic fetch routine. Microinstruction number 4 sets up the memory address register to fetch the second word of the instruction and microinstruction number 5 is used to bring this value from memory into the Am2903 ALU where it is added to the X2 register. The results of the addition are placed into the memory address register during this microinstruction. This value is used to fetch a value from memory which is placed in the Q register using microinstruction number 6. In the seventh microinstruction, the contents of the register pointed to by the X1 field are placed in the memory address register so that microinstruction eight can be utilized to bring this memory value into the Am2903s where it is added to the contents of the Q register with the result being placed into the Q register. Microinstruction number 9 is used to place this value back into the memory location as specified by the contents of the register pointed to by the X1 field. This memory address is still contained in the memory address register so that no updating is required. The total microcode required to implement this instruction routine is shown in Figure 10.

Register Immediate

The register immediate instruction is a very useful instruction which allows data to be added to the contents of the register. In this example, the second word of the instruction is fetched and added to the contents of the register specified in the R1 field.

Figure 11 depicts the microcode used to implement the register immediate instruction. Here, the first three microinstructions are the fetch routine for the instruction. The fourth microinstruction of this routine sets up the MAR to fetch the second word of the two word instruction. The contents of this memory location is brought into the Am2903 ALU and added to the contents of the register specified by the R1 field. The result of this operation is placed in the register specified by the R1 field.

Microinstruction Operation	Microcycle Time											
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	X												
Fetch Inst to IR		X											
Decode			X										
PC → MAR; PC + 1 → PC				X									
MEM + X2 → MAR					X								
MEM → Q						X							
(X1) → MAR							X						
MEM + Q → Q								X					
Q → MEM									X				

Figure 10. Memory to Memory Indexed Instruction Microcode.

Microinstruction Operation	Microcycle Time											
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	X												
Fetch Inst to IR		X											
Decode			X										
PC → MAR; PC + 1 → PC				X									
MEM + R1 → R1					X								

Figure 11. Register Immediate Instruction Microcode.

Memory Immediate

The memory immediate instruction is used to add immediate data contained in the second word of the instruction to a location in memory. The memory location is contained in the register specified in the X1 field of the instruction.

The memory immediate instruction is similar to the register immediate instruction except that an indirect addressing scheme is used. Again, the first three microinstructions fetch and decode the memory immediate instruction. The fourth and fifth microinstructions are used to fetch the data value which is the second word of this memory immediate instruction. Microinstruction number 4 sets up the memory address register and microinstruction number 5 brings the data into the Am2903 Q register. Microinstruction number 6 places the contents of the register specified by the X1 field into the memory address register so that the contents of this memory location can be brought into the Am2903 during microinstruction number 7. Here, during microinstruction 7 the contents of the Q register are added to this value and returned to the Q register. At microinstruction 8, the Q register is written back to the memory location as specified by the contents of the register pointed to by the X1 field. This value was already in the memory address register because it was used to fetch the operand originally at this location. The microcode for this instruction is detailed in Figure 12.

Improving Program Control Unit Performance

If we examine the microcode as shown for the various instruction types depicted in Figure 1, we find that all of these microroutines have several things in common. First, the very first microinstruction simply sets up the memory address register with the current value of the program counter. In addition, this microinstruction increments the current program counter value. The second microinstruction simply fetches the contents of memory and places it in the instruction register. The third microinstruction is used to decode the microinstruction, select the appropriate micromemory word and set it into the pipeline register. Finally, the fourth microinstruction begins actual execution of the desired instruction. In all of these examples and using the block diagram of Figure 4, we find that a bottle neck occurs in the ALU because of our need to be operating on program counter data and operand data intermixed. We can improve the performance of the program control unit by making the program counter an external register and using a multiplexer to select either the program counter or the Am2903 output to load the memory address register. This is depicted in block diagram form in Figure 13.

The first effect of implementing a program control unit with this architecture is that one of the instruction types is shortened by one microcycle. This is the register-to-memory-immediate instruction. The new microcode flowcharts for this instruction is

Microinstruction Operation	Microcycle Time												
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	X												
Fetch Inst to IR		X											
Decode			X										
PC → MAR; PC + 1 → PC				X									
MEM → Q					X								
(X1) → MAR						X							
MEM + Q → Q							X						
Q → MEM								X					

Figure 12. Memory Immediate Instruction Microcode.

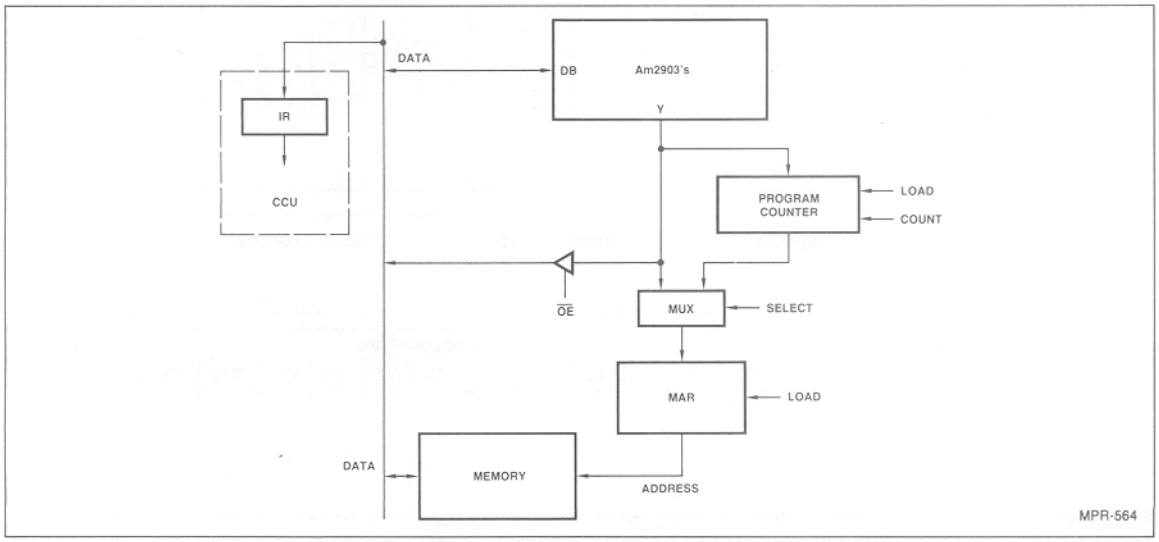

Figure 13. Memory Addressing Scheme with PC Outside of the ALU.

shown in Figure 14. In this case, we see that a PC value can be placed into the memory address register and the PC incremented while the ALU within the Am2903 is being used to perform either a pass or an addition. Thus, this architectural change has made some improvement in the thru-put of our machine.

The most important improvement in thru-put realized by the architecture shown in Figure 13 can be seen by evaluating the timing for sequential instructions. That is, what happens when several instructions are executed sequentially?

To keep the examples simple, let's visualize the microcycle timing chart for three register-to-register instructions executed sequentially. The most obvious timing chart would simply be to take the register-to-register microinstruction flows as shown in Figure 3 and concatenate three examples of this timing chart. If we do this, we will see that the final execution of the values of R1 + R2 return to R1 utilize the ALU, but the program counter is not in operation. However, the next microcycle requires placing the program counter into the memory address register. Thus, the architecture of Figure 13 allows us to do these two micro-operations during the same microinstruction. If we assume three register-to-register instructions in sequence in memory; let's call them instruction A, B and C; the timing chart of Figure 15 results. What we see in this diagram is that the execution of instruction A can be overlapped with the set up the program counter in memory address register for fetching instruction B. Thus, instead of instruction B starting at time T4, it may be started at time T3. This can be accomplished by simply having the execution microinstruction also load the MAR with the current PC value and increment the PC. From this discussion, we can see that instead of twelve microcycle times being required to execute three register-to-register instructions, only nine microcycle times will be required. We should caution that if the reader counts the microcycles in Figure 15, he will arrive at 10 microcycle times being required. This leads us to our next point.

If we examine all of the instructions described earlier in this application note, we will find that in all cases, the execution of the instruction (the last microcycle) can be overlapped with the first microinstruction of the fetch routine. Thus, the architectural change shown in Figure 13 not only allows three of the instructions to execute faster during their total microcode, but in fact all microinstructions can be executed at least one microcycle faster because of the ability to overlap the first microcycle of the fetch routine with the execution of the instruction. This architectural change therefore saves one or two microcycles depending on the instruction.

In Chapter 9 we will show how further overlapping at the machine instruction level can allow us to execute a register-to-register instruction during every microcycle, effectively; rather than every three microcycles as shown in Figure 15. At the present time, let us simply leave the discussion at this point.

Subroutining

An implementation technique that is common to the different addressing modes is the subroutine (also called stack and link). The subroutine allows sections of main program to access a common subsection of the program. The general effect is to allow less lines of machine code to be written for any given program that employs subroutines.

Figure 16 shows an example of a subroutine within the program. The main program executes instructions until it gets to instruction 52 which is a call to subroutine. This instruction puts address 80 in the program counter while saving address 53 in a separate register called Return Register. The program continues on from address 80 to address 85 where it encounters the return from subroutine command. The return-from-subroutine command takes a value out of the return register and puts that into the program counter. At that point the program counter continues down in the main body of the program until it reaches address 57. At this time, another call to subroutine may occur forcing the program counter back to the value of 80 while putting the value 58 into the return address. The subroutine is executed and at address 85 the return command is again encountered. At this point,

Microinstruction Operation	Microcycle Time												
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	X												
Fetch Inst to IR		X											
Decode			X										
(X2) → MAR				X									
MEM + R1 → R1					X								
PC → MAR; PC + 1 → PC					X								
MEM + R1 → R1						X							

Figure 14. Register to Memory Immediate Instruction Improved Microcode.

Microinstruction Operation	Microcycle Time												
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	A			B			C						
Fetch Inst to IR		A			B			C					
Decode			A			B			C				
R1 + R2 → R1				A			B			C			

Figure 15. Register to Register Instruction with Overlap of Execute and PC Control.

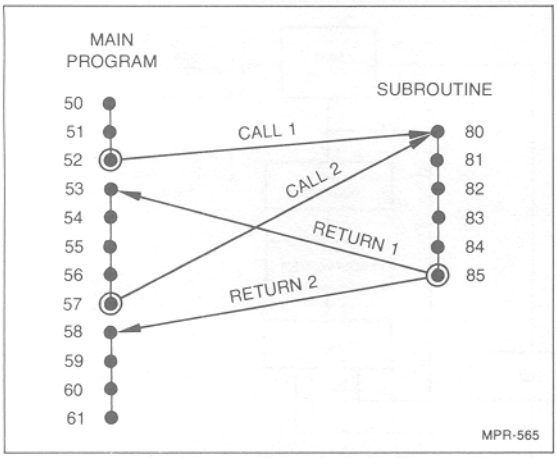

Figure 16. Subroutine Execution.

the subroutine will return control of the program to address 58 of the instruction stream and the main program continues to sequence through its instructions.

In many systems, one subroutine may very well call another subroutine which may in turn call yet another subroutine and so on. To accomplish this the return address linkage must now be "nested" using a last-in first-out (LIFO) stacking arrangement. Figure 17 illustrates subroutine nesting. In this example, the main program contains a subroutine call or jump-to-subroutine command (JSB) at address 53. Program control is passed to the first subroutine at address 88, while the return address 54 is placed in the stack. At address 89 the of the subroutine 1 another JSB command is encountered passing the program control to Subroutine 2 at address 502. The return address value 90 is pushed onto the top of the stack. This continues in like fashion for calls to Subroutine 3 and 4 with return address 506 and 723 being placed on the stack. At address 785 of Subroutine 4, a Return from Subroutine (RTS) command is decoded causing the return address 723 on the top of the stack to be placed in the program counter and the contents of the stack are "poped" up one place.

At address 725 another RTS command is found, causing the top of the stack, address 506, to be placed in the program counter and the stack is poped. The identical action occurs for the RTS commands at address 507 and 92 such that control is eventually returned to the main program and the stack is empty.

The LIFO or subroutine stack in the program control hardware is shown in Figure 18. When the call from subroutine command is decoded by the computer control unit, the pipeline register outputs cause the stack control to accept the output of the program counter register and place it at the top of the stack. Next the subroutine address is brought in from the memory passed through the multiplexer and placed in the MAR. The subroutine address is also brought through the multiplexer incrementer, through the incrementer and placed in the program counter register to be used as a possible next source of address. The subroutine return address is recovered from the stack when the pipeline register instructs the stack control logic to place the return address at the multiplexer. The return address is passed through the multiplexer and clocked into the MAR. The return address is also clocked into the PC register via the incrementer multiplexer and the incrementer, for use as the next sequential address. Figure 19 shows the jump to subroutine instruction and Figure 20 shows the microcycles that are used in a typical call to subroutine command using the program control hardware shown in Figure 18. At T0 the program counter is placed into the MAR and updated. Time T1 finds the MAR accessing the subroutine call instruction, with the instruction being placed into the instruction register. At T2 the opcode is decoded by the CCU, and the first instruction microcode bits are clocked into the pipeline register. At time T3, the PC is placed in the MAR. At T4 the starting address of the subroutine is being fetched and placed into the MAR; the stack pointer is incremented; the current program counter is placed on the LIFO stack; and the starting address of the Subroutine plus one is placed into the program counter.

Figure 21 details the microcycle timing for a return-from-subroutine execution. At time zero the current program counter is placed into the MAR, then incremented by one. During time one the contents of the MAR fetches the return from subroutine command, which is then clocked into the instruction register at the end of the microcycle. At time 2 the contents of the instruction register is decoded in the CCU with the control bits being clocked into the pipeline register. During time 3 the return address on the top of

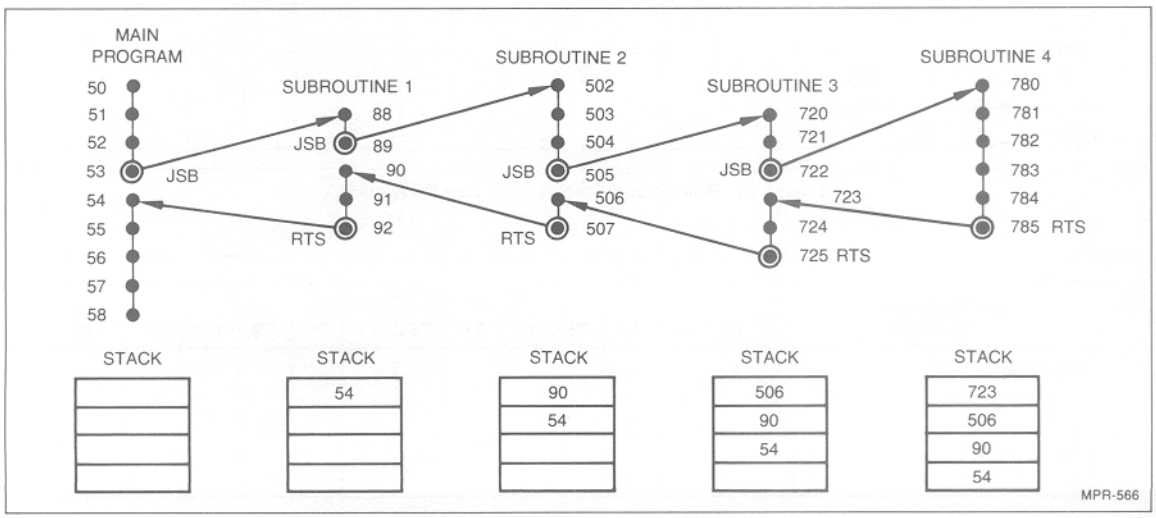

Figure 17. Nested Subroutine Example.

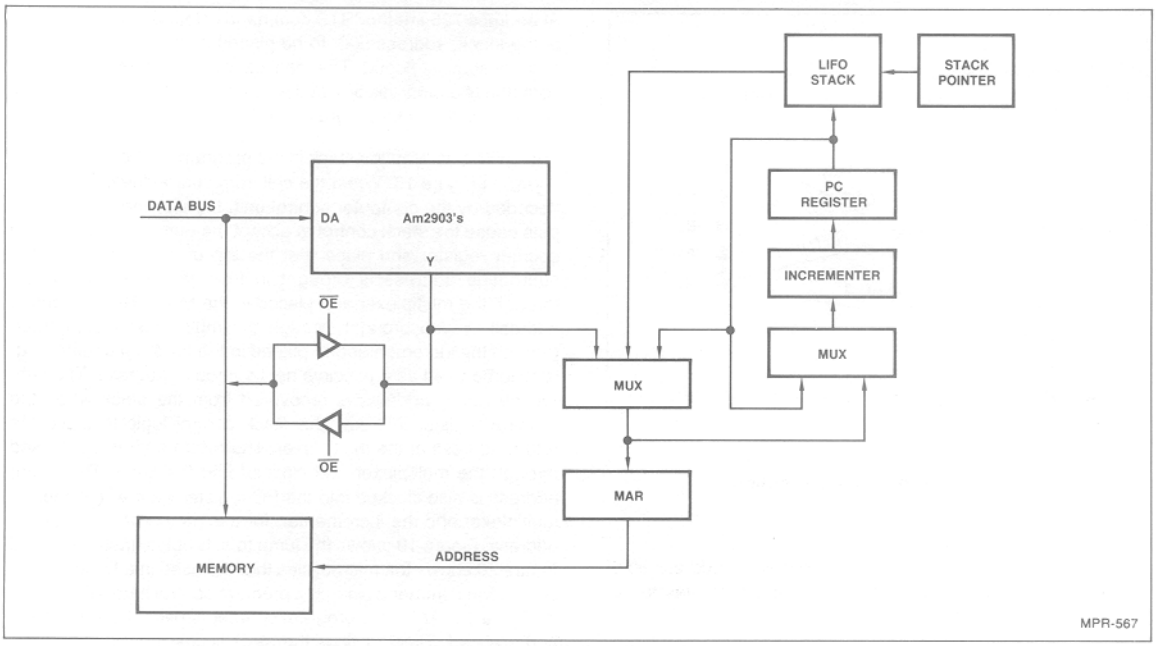

Figure 18. Subroutine Stack Architecture.

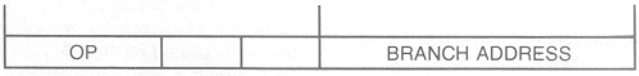

Figure 19. Jump to Subroutine (Branch and Stack) Instruction.

Microinstruction Operation	Microcycle Time T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	X												
Fetch Inst to IR		X											
Decode			X										
PC → MAR; PC + 1 → PC				X									
MEM → MAR; PC → STACK MEM + 1 → PC; SP + 1 → SP					X								

Figure 20. Branch and Stack Instruction Microcode.

Microinstruction Operation	Microcycle Time T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
PC → MAR; PC + 1 → PC	X												
Fetch Inst to IR		X											
Decode			X										
Stack → MAR; Stack + 1 → SP SP − 1 → SP				X									

Figure 21. Return from Subroutine Instruction Microcode.

the LIFO stack is placed into the MAR, while that value plus one is stored into program counter. The stack pointer is then decremented.

The basic program control hardware thus developed with some embellishments added are contained within the Am2930 program control unit as shown in Figure 22. The Am2930 is a 4-bit slice of the program control unit. It therefore easily allows the address bus to be virtually independent of the data bus in terms of width. The Am2930 has a general purpose auxiliary register which has two sources and two destinations. One source being the D inputs which flow through the R multiplexer and hence into the auxiliary register and the other source being the output of the full adder which is the second input to the R multiplexer. The two outputs of the auxiliary register go to the A and B multiplexers which in turn source the A and B inputs to the full adder. The register enable pin ($\overline{RE}$) allows the auxiliary register to be unconditionally loaded from the D Inputs of the Am2930. The A multiplexer selects as its sources a logical zero, the output of the auxiliary register, or the D inputs. The B multiplexer accepts the outputs of the auxiliary register, a logical zero, the output of the subroutine stack file, or the output of the program counter register as its sources.

In the Am2930 design the LIFO stack is 17 words deep, allowing up to seventeen levels of subroutine. The LIFO stack is controlled by the stack pointer logic which gives a FULL indication when the stack is full and an EMPTY indication when the stack has emptied. The input to the LIFO stack is fed through a stack multiplexer whose inputs may be D inputs or the output of the program counter. Thus, depending upon the application, the stack may be used as either a subroutine stack or a general purpose LIFO stack which resides on the D bus. The incrementer and the full adder are controlled by the Ci and Cn carry-in bits respectively. Figure 23 details the ripple carry connections between Am2930s in a 16-bit array. The Ci input of the least significant slice (LSS) is controlled from the pipeline register.

The Ci signal is internally propagated through the incrementer of each device using carry look ahead logic. The microprogram memory, using the Ci input may now cause the Am2930s to repeatedly access the same main memory instruction if so desired. The full adder has its Cn input tied to ground for the LSS device of the Am2930 array. The Cn signal is progagated in parallel through the Am2930s.

For a faster propagation of the Cn signal the interconnection shown in Figure 24 should be employed. The generate and propagate pins ($\overline{G}$, $\overline{P}$) of the Am2902A carry look ahead generator. The look ahead carries (Cn + x, y, z) are connected to the Cn inputs of their respective devices. The output of the Am2930 is three-state and is controlled by the output enable pin

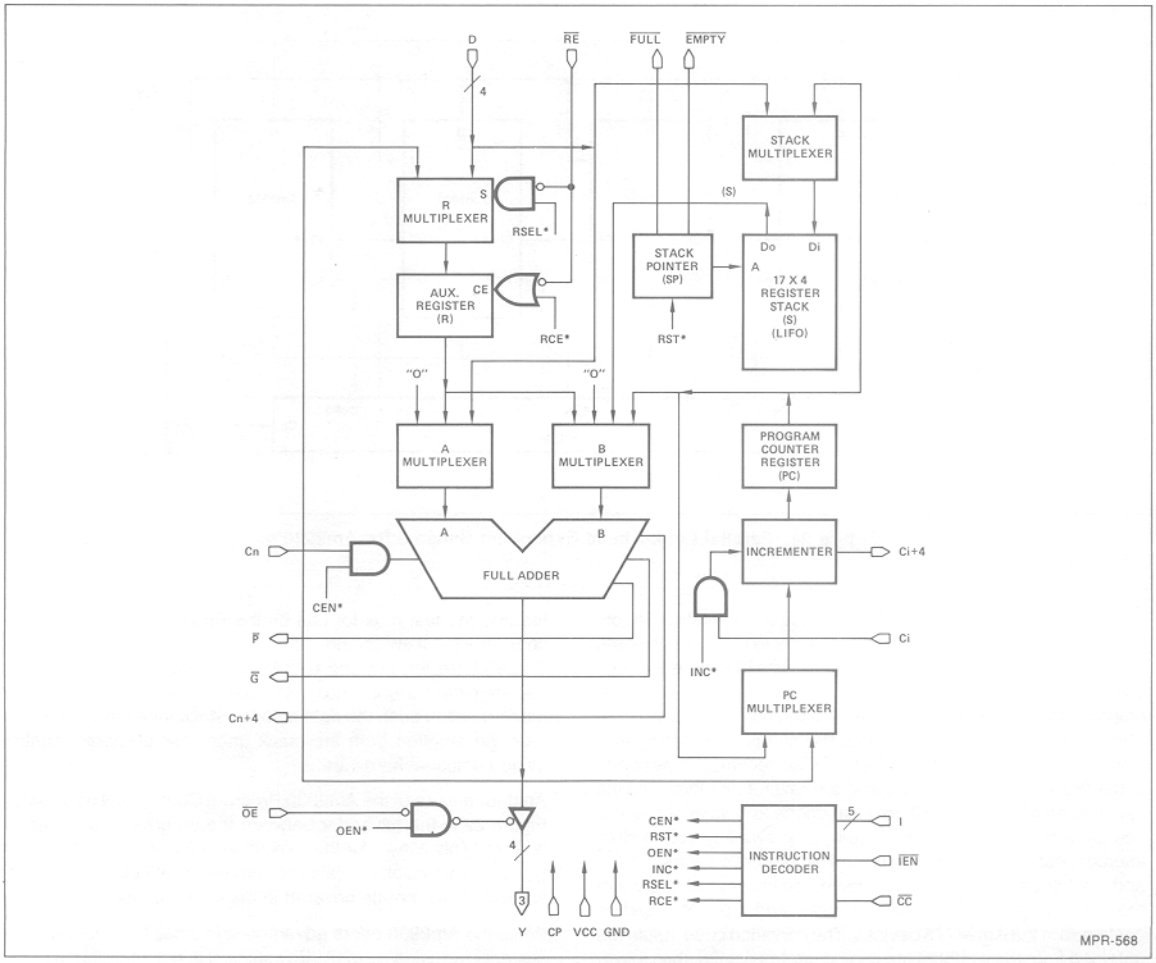

Figure 22. Am2930 Block Diagram.

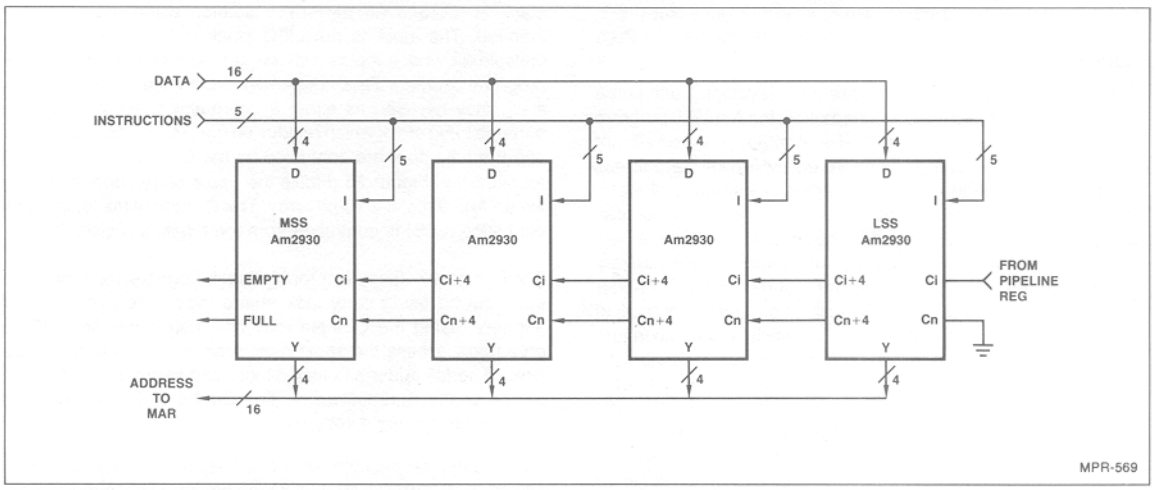

Figure 23. Ripple Expansion Scheme for Am2930's.

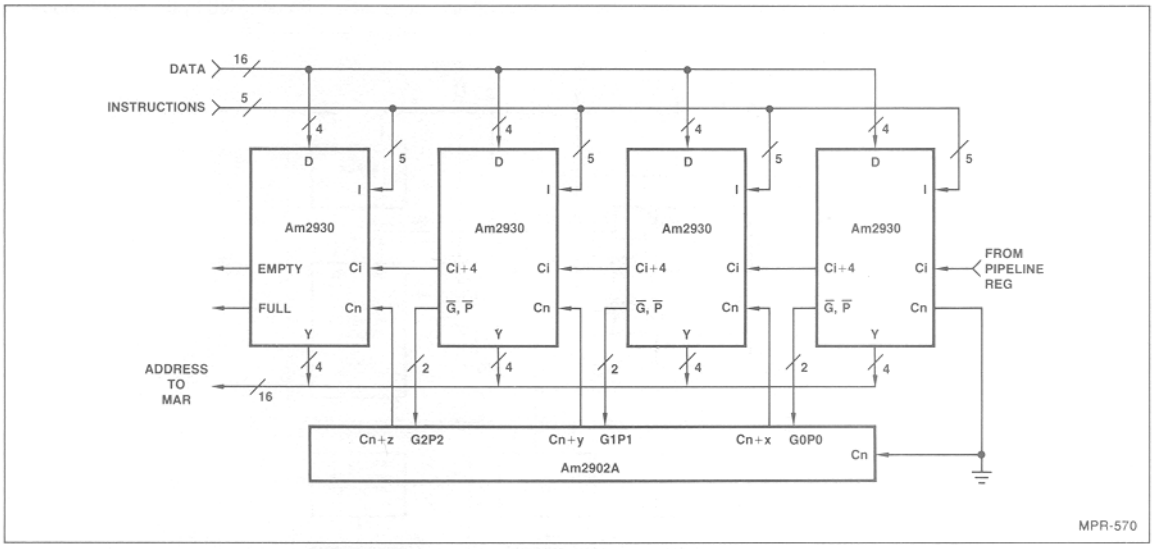

Figure 24. Parallel Look-Ahead Expansion Scheme for Am2930's.

($\overline{OE}$). Other features of the Am2930 include an Instruction Enable pin ($\overline{IEN}$). This pin allows the Am2930 array to be taken off of the microprogram data bus thus allowing the bits that were formerly committed to the Am2930 to be used in conjunction with other devices. The Am2930 also includes a condition code input ($\overline{CC}$). The Condition Code input permits the conditional testing of a single bit. This allows the feasibility of such techniques as conditional branching at the macroprogram level. For more detailed explanation of the Am2930, its instructions and its applications, see the Am2930 Data Sheet. Figure 25 shows a typical system interconnection using the Am2930. The instruction lines, Ci, $\overline{RE}$ and the $\overline{OE}$ control pins are connected directly to the outputs of the combination microprogram memory and pipeline registers contained in the Am24775 devices. The condition code inputs are obtained from the Am2904 status and control device, thus allowing conditional jumps on status. Status from the Am2904 is also fed into the test mux for use by the Am2910 for its conditional code input. Likewise the full and empty indications from the Am2930 are fed into the test MUX for use by the Am2910 to ascertain the current status of the stack. If the stack is full and the user wishes to push the data onto the stack then the current data must be emptied from the stack under microprogram control, using additional hardware.

Another feature of the Am2930 Program Control Unit as shown in Figure 22 is the full adder between the program counter and Y outputs. This allows for the execution of PC relative addressing types of instructions. While this can be an effective addressing scheme, it will not be covered in detail in this application note.

While the Am2930 offers advantages in small high performance systems requiring a small LIFO stack, it is not intended to be the solution for all program counter requirements.

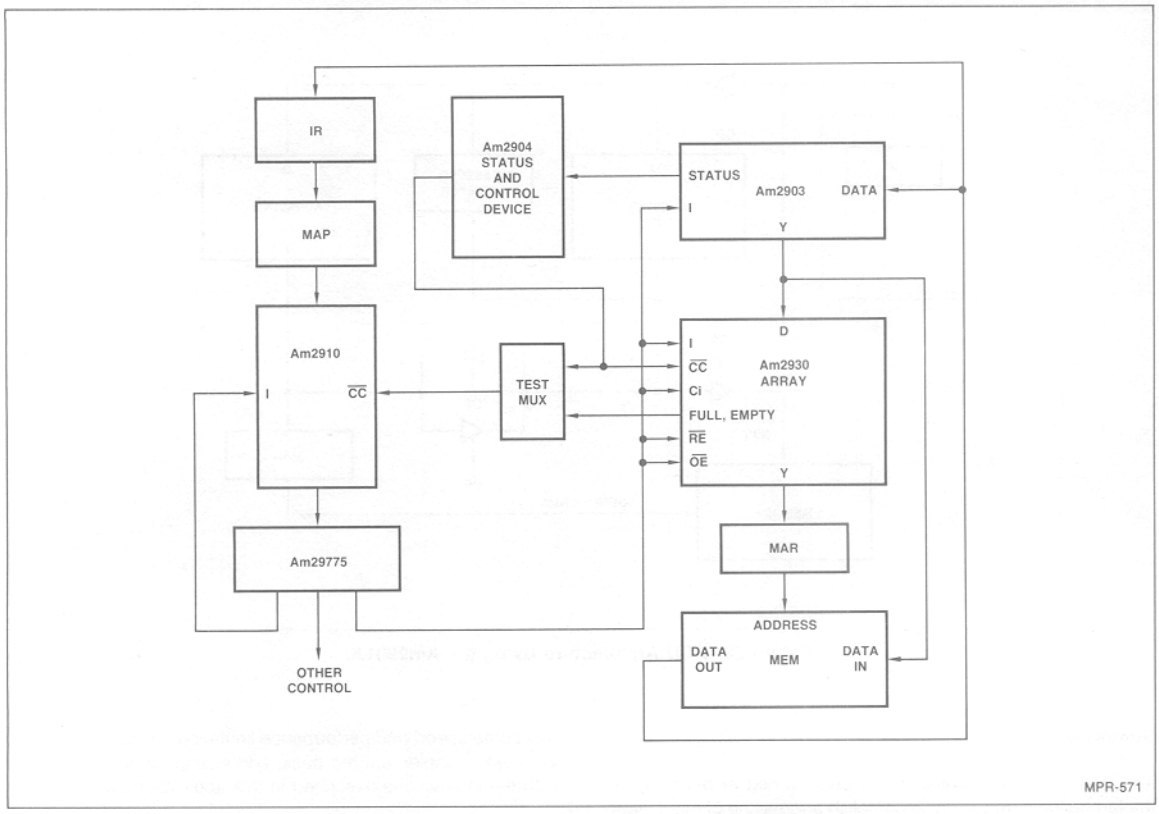

Figure 25. System Interconnection Using the Am2930.

Using the Am2901A as a Program Control Unit

Up to this point, the discussion has concerned a general architecture which includes 16 general registers in the ALU section and the LIFO stack is a program control section as shown in Figure 18. An alternative architecture and that used by most general purpose machines, is to place the LIFO stack in main memory. The stack pointer for the main memory LIFO stack can be contained in the program control unit to be described in this section. If the program control unit is built using Am2901A's it now has the capability of using its internal registers as the program counter, stack pointer, upper stack bound pointer, lower stack bound pointer, and internal temporary registers. This of course provides considerable flexibility in the architecture and also allows for a much greater repertoire of instructions to be executed. Particularly, several stack instructions can be included in the instruction set, most of which will use the form of the register-to-indexed-memory instruction format as shown in Figure 1.

Another advantage of the architecture shown in Figure 25 is speed. The Am2901A's slightly surpass the Am2903 in speed. Thus, a 16-bit Am2901A program control unit architecture can be implemented and it will perform well within the microcycle times budgeted for the system.

Looking at Figure 26 which shows the Am2901A used as a program control unit and the Am2903 used for the general register stacks/ALU section, we see a three-state buffer on the Y outputs of the Am2903 connected to the data bus as well as a three-state buffer at the input of the Am2903's from the data bus. This provides isolation and buffering for the bus as well as allowing appropriate disconnects so that certain microcycles can be combined to improve the overall performance of the machine. In addition a transfer register is used between the Am2903's and Am2901s to allow a microcycle to be terminated if an ALU operation is taking place within the Am2903's. This provides higher performance operation for the machine. In addition, a bi-directional buffer (such as the Am8304B) is used between the Am2901A Y-outputs and the Am2903 Y-outputs. This gives the ability to push the program counter contained in the Am2901A on the stack for interrupt handling. In addition, values coming from the Am2903 can be placed in the memory address register.

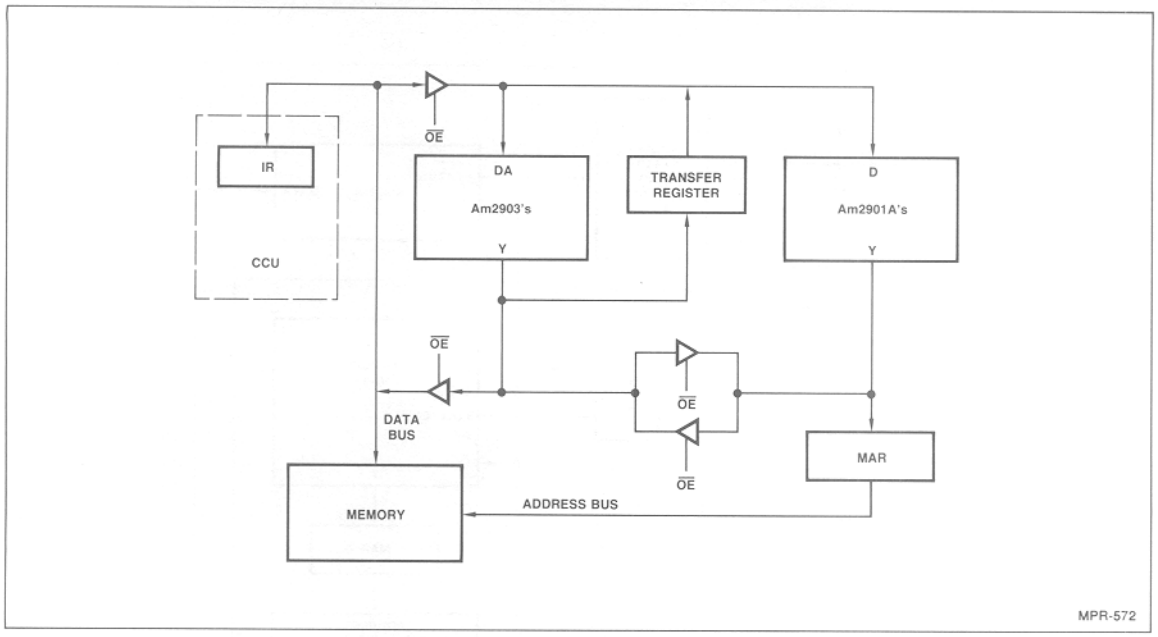

Figure 26. PCU Architecture Using the Am2901A.

Summary

The thrust of this discussion has been aimed at defining and implementing hardware to accomplish addressing of main memory. We have shown that a speed advantage is realized if the program counter is kept separate from the main general purpose register stack/ALU hardware. The most general purpose program control unit is the Am2901A. It offers several advantages in terms of program control, stack pointer control, and stack pointer boundary conditions. The Am2930 can be used in program control units occupying less space and including a built-in stack, but has some speed and performance limitations. Both devices can be used to implement the basic addressing modes associated with the instructions described in this application note.

Another purpose of this application note is to set the stage for Chapter 9 where we will overlap machine instructions such that register to register instructions can be executed in a single 200ns microcycle and the memory reference instructions can be executed in 600ns (3 microcycles) as the effective execution time. Also, we will expand on the use of the Am2901A as a Program Control Unit.

ADVANCED
MICRO
DEVICES, INC.
901 Thompson Place
Sunnyvale
California 94086
(408) 732-2400
TWX: 910-339-9280
TELEX: 34-6306
TOLL FREE
(800) 538-8450

11-78