

Chapter 13

Bit-Sliced Microprocessor of the Am2900 Family: The Am2901/2909¹

Introduction

The Am2900 Family

The Am2900 Family consists of a series of LSI building blocks designed for use in microprogrammed computers and controllers. Each device is designed to be expandable and sufficiently flexible to be suitable for emulation of many existing machines.

Figure 1 illustrates a typical system architecture. There are two "sides" to the system. At the left is the control circuitry and on the right is the data manipulation circuitry. The block labeled "2901 array" consists of the ALU, scratchpad registers, and data steering logic (all internal to the Am2901's), plus left/right shift control and carry lookahead circuit. Data is processed by moving it from main memory (not shown) into the 2901 registers, performing the required operations on it, and returning the result to main memory. Memory addresses may also be generated in the 2901's and sent out to the memory address register (MAR). The four status bits from the 2901's ALU are captured in the status register after each operation.

The logic on the left side is the control section of the computer. This is where the Am2909 is used. The entire system is controlled by a memory, usually PROM, which contains long words called microinstructions. Each microinstruction contains bits to control each of the data manipulation elements in the system. There are, for example, 9 bits for the 2901 instruction lines, 8 bits for the A and B register addresses, 2 or 3 bits to control the shifting multiplexers at the ends of the 2901 array, and bits to control the register enables on the MAR, instruction register, and various bus transceivers. When the bits in a microinstruction are applied to all the data elements and everything is clocked, then one small operation (such as a data transfer or a register-to-register add) will occur.

Each microinstruction contains not only bits to control the data hardware, but also bits to define the location in PROM of the next microinstruction to be executed. The fields are labeled in Fig. 1 as I, CC, and BA. The I field controls the sequencer. It indicates where the next address is located—the μ PC, the stack, or the direct inputs—and whether the stack is to be pushed or popped.

The CC field contains bits indicating the conditions under which the I field applies. These are compared with the condition codes in the status register and may cause modification to the I field. The comparing and modification occurs in the block labeled "control logic." Frequently this is just a PROM. The BA field is a branch address or the address of a subroutine.

Pipelining

The address for the microinstructions is generated by the sequencer, starting from a clock edge. The address goes from the sequencer to the ROM, and an access time later, the microinstruction is at the ROM outputs.

A pipeline register is a register placed on the output of the microprogram memory to essentially split the system in two. The pipeline register contains the microinstruction currently being executed ①. (Refer to the circled numbers in Fig. 1.) The data manipulation control bits go out to the system elements and a portion of the microinstruction is returned to the sequencer ② to determine the address of the next microinstruction to be executed. That address ③ is sent to the ROM, and the next microinstruction ④ sits at the input of the pipeline register. So while the 2901's are executing one instruction, the next instruction is being fetched from ROM. Note that there is no sequential logic in the sequencer between the select lines and the output. This is important because the loop ① to ② to ③ to ④ must occur during a single clock cycle. During the same time, the loop from ① to ⑤ must occur in the 2901's. These two paths are roughly the same (around 200 ns worst case for a 16-bit system). The presence of the pipeline register allows the microinstruction fetch to occur in parallel with the data operation rather than serially, allowing the clock frequency to be doubled.

The emulation of an existing machine by Fig. 1 works as follows. A sequence of microinstructions in the PROM is executed to fetch an instruction from main memory. This requires that the program counter, often in a 2901 working register, be sent to the memory address register and incremented. The data returned from memory is loaded into the instruction register. The contents of the instruction register are passed through a PROM or PLA to generate the address of the first microinstruction which must be executed to perform the required function. A branch to this address occurs through the sequencer. Several microinstructions may be executed to fetch data from memory, perform ALU operations, test for overflow, and so forth. Then a branch will be made back to the instruction fetch cycle. At this point, there may be branches to other sections of microcode. For example, the machine might test for an interrupt here and obtain an interrupt service routine address from another mapping ROM rather than start on the next machine instruction.

¹Abstracted from *The Am2900 Family Data Book*, Advanced Micro Devices, Inc., 1976.

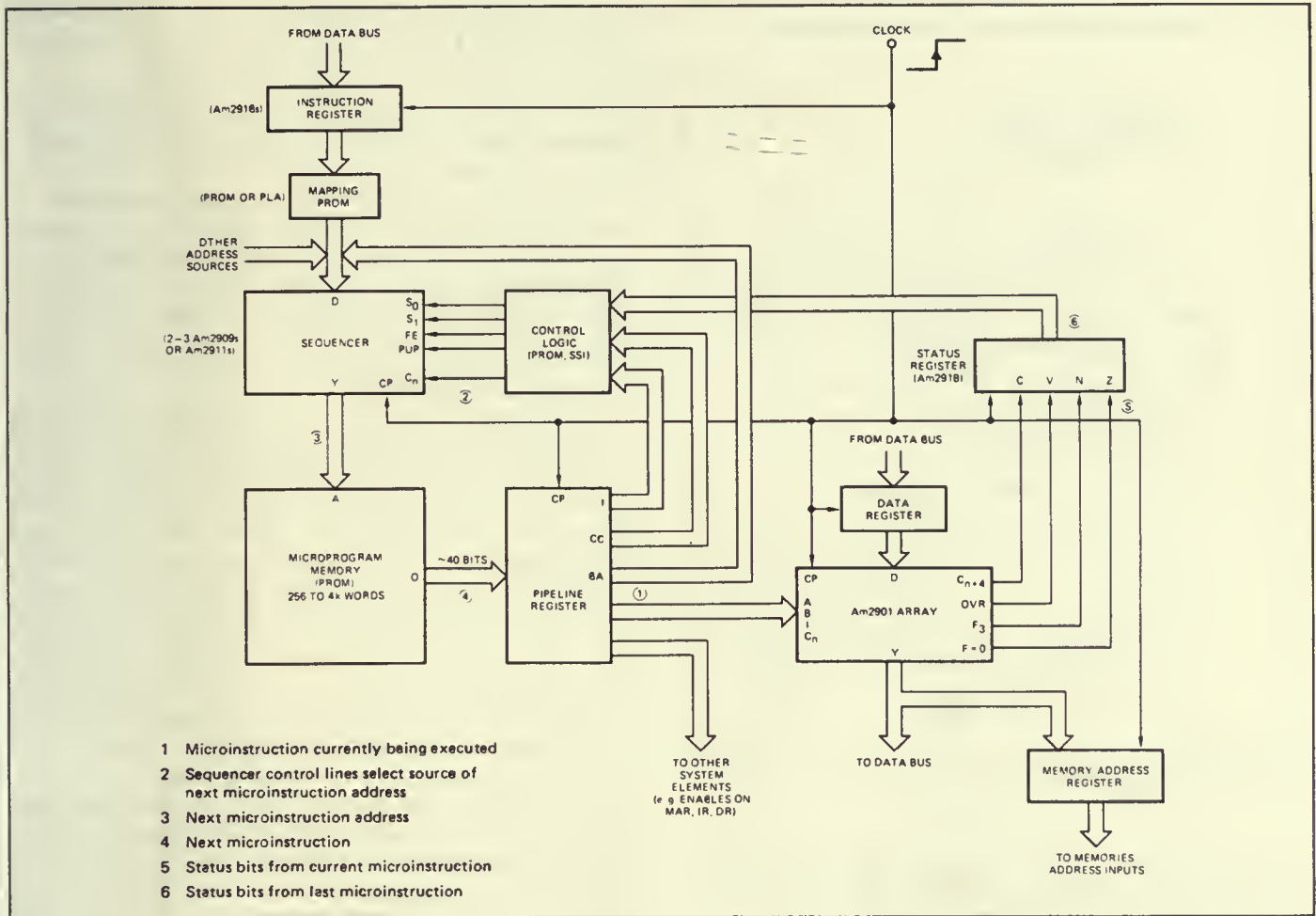


Fig. 1

Am2901: Four-Bit Bipolar Microprocessor Slice

The device, as shown in Fig. 2, consists of a 16-word by 4-bit two-port RAM, a high-speed ALU, and the associated shifting, decoding, and multiplexing circuitry. The 9-bit microinstruction word is organized into three groups of 3 bits each and selects the ALU source operands, the ALU function, and the ALU destination register. The microprocessor is cascadable with full lookahead or with ripple carry, has three-state outputs, and provides various status flag outputs from the ALU. Advanced low-power Schottky processing is used to fabricate this 40-lead LSI chip.

Architecture

A detailed block diagram of the bipolar microprogrammable microprocessor structure is shown in Fig. 3. The circuit is a 4-bit slice cascadable to any number of bits. Therefore, all data paths within the circuit are 4 bits wide. The two key elements in the Fig. 3 block diagram are the 16-word by 4-bit two-port RAM and the high-speed ALU.

Data in any of the 16 words of the random-access memory (RAM) can be read from the A port of the RAM as controlled by the 4-bit A address field input. Likewise, data in any of the 16 words of the RAM as defined by the B address field input can be

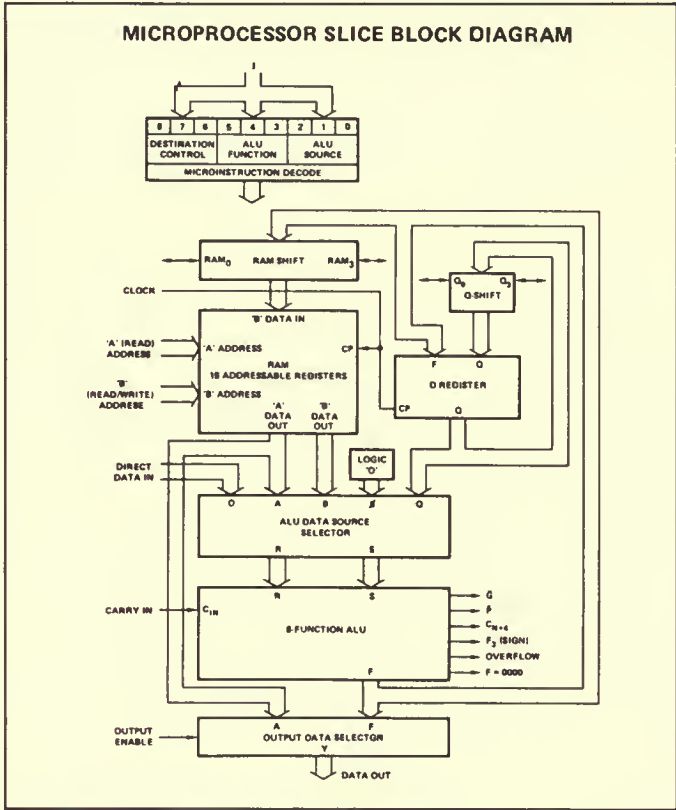


Fig. 2. Microprocessor slice block diagram.

simultaneously read from the B port of the RAM. The same code can be applied to the A select field and B select field, in which case the identical file data will appear at both the RAM A port and B port outputs simultaneously.

When enabled by the RAM write enable (RAM EN), new data is always written into the field (word) defined by the B address field of the RAM. The RAM data-input field is driven by a three-input multiplexer. This configuration is used to shift the ALU output data (F) if desired. This three-input multiplexer scheme allows the data to be shifted up one bit position, shifted down one bit position, or not shifted in either direction.

The RAM A port data outputs and RAM B port data outputs drive separate 4-bit latches. These latches hold the RAM data while the clock input is LOW. This eliminates any possible race conditions that could occur while new data is being written into the RAM.

The high-speed Arithmetic Logic Unit (ALU) can perform three binary arithmetic and five logic operations on the two 4-bit words R and S. The R input field is driven from a two-input multiplexer,

while the S input field is driven from a three-input multiplexer. Both multiplexers also have an inhibit capability; that is, no data is passed. This is equivalent to a zero source operand.

In Fig. 3, the ALU R-input multiplexer has the RAM A port and the direct data inputs (D) connected as inputs. Likewise, the ALU S-input multiplexer has the RAM A port, the RAM B port, and the Q register connected as inputs.

The two source operands not fully described as yet are the D input and Q input. The D input is the 4-bit-wide direct data-field input. This port is used to insert all data into the working registers inside the device. Likewise, this input can be used in the ALU to modify any of the internal data files. The Q register is a separate 4-bit file intended primarily for multiplication and division routines, but it can also be used as an accumulator or holding register for some applications.

This multiplexer scheme gives the capability of selecting various pairs of the A, B, D, Q, and O inputs as source operands to the ALU. These five inputs, when taken two at a time, result in ten possible combinations of source operand pairs. These combinations include AB, AD, AQ, AO, BD, BQ, BO, DQ, DO, and QO. It is apparent that AD, AQ, and AO are somewhat redundant with BD, BQ, and BO in that if the A address and B address are the same, the identical function results. Thus, there are only seven completely non-redundant source operand pairs for the ALU. The Am2901 microprocessor implements eight of these pairs. The microinstruction inputs used to select the ALU source operands are the I₀, I₁, and I₂ inputs. The definitions of I₀, I₁, and I₂ for the eight source operand combinations are as shown in Table 1. Also shown is the octal code for each selection.

The I₃, I₄, and I₅ microinstruction inputs are used to select the ALU function. The definition of these inputs is shown in Table 2. The octal code is also shown for reference. The normal technique for cascading the ALU of several devices is in a lookahead carry mode. Carry generate, \overline{G} , and carry propagate, \overline{P} , are outputs of the device for use with a carry-lookahead generator such as the

Table 1 ALU Source Operand Control

Microcode				ALU source operands	
Octal code				R	S
I ₂	I ₁	I ₀			
L	L	L	0	A	Q
L	L	H	1	A	B
L	H	L	2	O	Q
L	H	H	3	O	B
H	L	L	4	O	A
H	L	H	5	D	A
H	H	L	6	D	Q
H	H	H	7	D	O

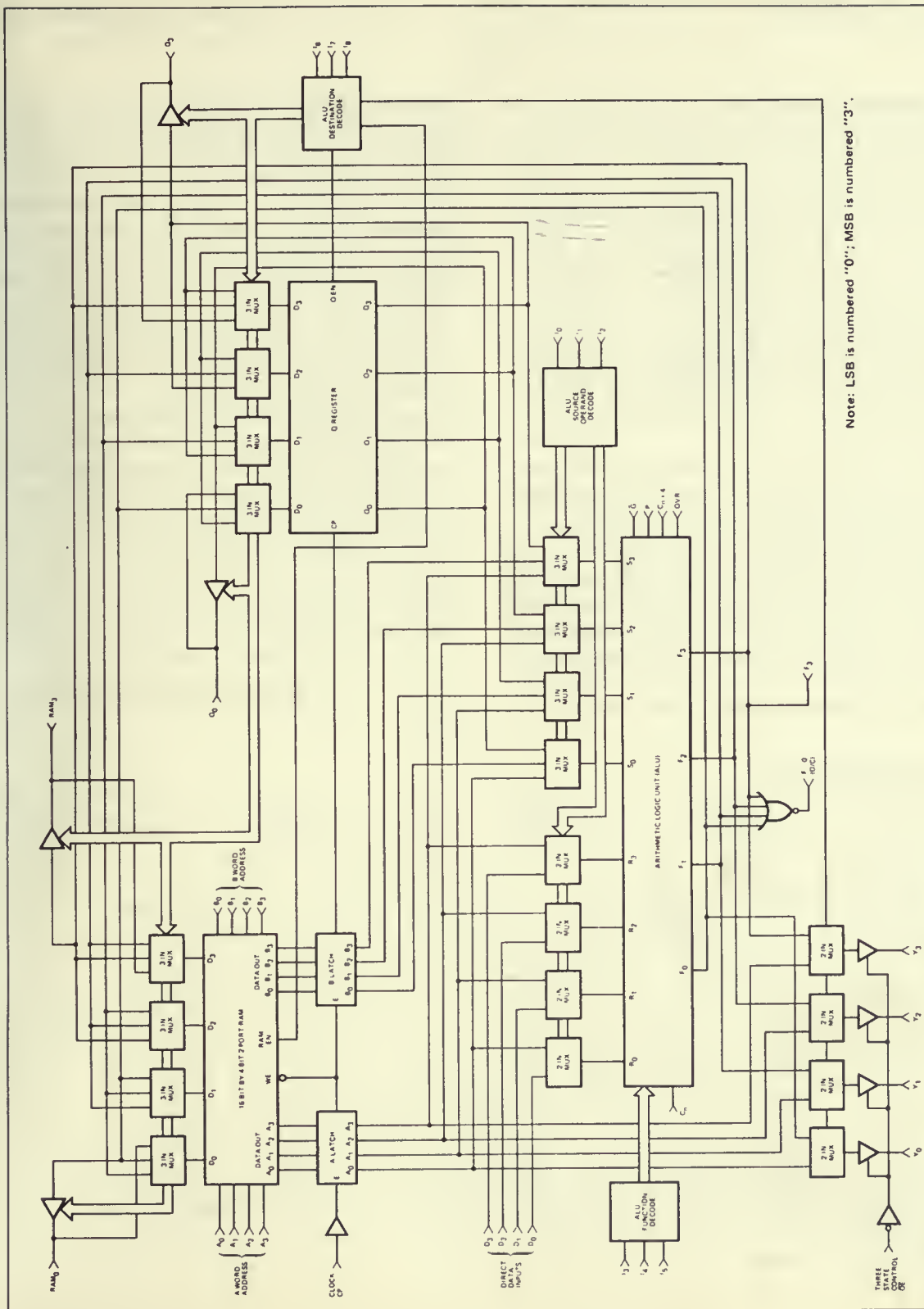


Fig. 3. Detailed Am2901 microprocessor block diagram.

Table 2 ALU Function Control

Microcode				ALU function	Symbol
<i>I</i> ₅	<i>I</i> ₄	<i>I</i> ₃	Octal code		
L	L	L	0	R plus S	R + S
L	L	H	1	S minus R	S − R
L	H	L	2	R minus S	R − S
L	H	H	3	R OR S	R ∨ S
H	L	L	4	R AND S	R ∧ S
H	L	H	5	\bar{R} AND S	$\bar{R} \wedge S$
H	H	L	6	R EX-OR S	$R \nabla S$
H	H	H	7	R EX-NOR S	$R \nabla \bar{S}$

Am2902 ('182). A carry-out, C_{n+4} , is also generated and is available as an output for use as the carry flag in a status register. Both carry-in (C_n) and carry-out (C_{n+4}) are active HIGH.

The ALU has three other status-oriented outputs. These are F_3 , $F = 0$, and overflow (OVR). The F_3 output is the most significant (sign) bit of the ALU and can be used to determine positive or negative results without enabling the three-state data outputs. F_3 is non-inverted with respect to the sign bit output Y_3 . The $F = 0$ output is used for zero detect. It is an open-collector output and can be wire ORed between microprocessor slices. $F = 0$ is HIGH when all F outputs are LOW. The overflow output

(OVR) is used to flag arithmetic operations that exceed the available 2's complement number range. The overflow output (OVR) is HIGH when overflow exists; that is, when C_{n+3} and C_{n+4} are not the same polarity.

The ALU data output is routed to several destinations. It can be a data output of the device and it can also be stored in the RAM or the Q register. Eight possible combinations of ALU destination functions are available as defined by the I_6 , I_7 , and I_8 microinstruction inputs. These combinations are shown in Table 3.

The 4-bit data output field (Y) features three-state outputs and can be directly bus-organized. An output control (\overline{OE}) is used to enable the three-state outputs. When \overline{OE} is HIGH, the Y outputs are in the high-impedance state.

A two-input multiplexer is also used at the data output such that either the A port of the RAM or the ALU outputs (F) are selected at the device Y outputs. This selection is controlled by the I_6 , I_7 , and I_8 microinstruction inputs. Refer to Table 3 for the selected output for each microinstruction code combination.

As was discussed previously, the RAM inputs are driven from a three-input multiplexer. This allows the ALU outputs to be entered non-shifted, shifted up one position (multiplied by 2), or shifted down one position (divided by 2). The shifter has two ports; one is labeled RAM_0 and the other is labeled RAM_3 . Both of these ports consist of a buffer-driver with a three-state output and an input to the multiplexer. Thus, in the shift-up mode, the RAM_3 buffer is enabled and the RAM_0 multiplexer input is

Table 3 ALU Destination Control

Microcode				RAM function		Q-register function		Y output	RAM shifter		Q shifter	
<i>I</i> ₈	<i>I</i> ₇	<i>I</i> ₆	Octal Code	Shift	Load	Shift	Load		<i>RAM</i> ₀	<i>RAM</i> ₃	<i>Q</i> ₀	<i>Q</i> ₃
L	L	L	0	X	None	None	F → Q	F	X	X	X	X
L	L	H	1	X	None	X	None	F	X	X	X	X
L	H	L	2	None	F → B	X	None	A	X	X	X	X
L	H	H	3	None	F → B	X	None	F	X	X	X	X
H	L	L	4	Down	F/2 → B	Down	Q/2 → Q	F	<i>F</i> ₀	<i>IN</i> ₃	<i>Q</i> ₀	<i>IN</i> ₃
H	L	H	5	Down	F/2 → B	X	None	F	<i>F</i> ₀	<i>IN</i> ₃	<i>Q</i> ₀	X
H	H	L	6	Up	2F → B	Up	2Q → Q	F	<i>IN</i> ₀	<i>F</i> ₃	<i>IN</i> ₀	<i>Q</i> ₃
H	H	H	7	Up	2F → B	X	None	F	<i>IN</i> ₀	<i>F</i> ₃	X	<i>Q</i> ₃

X—Don't care. Electrically, the shift pin is a TTL input internally connected to a three-state output which is in the high impedance state.

B—Register Addressed by B inputs.

Up is toward MSB. Down is toward LSB.

enabled. Likewise, in the shift-down mode, the RAM_0 buffer and RAM_3 input are enabled. In the no-shift mode, both buffers are in the high-impedance state and the multiplexer inputs are not selected. This shifter is controlled from the I_6 , I_7 , and I_8 microinstruction inputs as defined in Table 3.

Similarly, the Q register is driven from a three-input multiplexer. In the no-shift mode, the multiplexer enters the ALU data into the Q register. In either the shift-up or shift-down mode, the multiplexer selects the Q register data appropriately shifted up or down. The Q shifter also has two ports; one is labeled Q_0 and the other is Q_3 . The operation of these two ports is similar to the RAM shifter and is also controlled from I_6 , I_7 , and I_8 as shown in Table 3.

The clock input to the Am2901 controls the RAM, the Q register, and the A and B data latches. When enabled, data is clocked into the Q register on the LOW-to-HIGH transition of the clock. When the clock input is HIGH, the A and B latches are open and will pass whatever data is present at the RAM outputs. When the clock input is LOW, the latches are closed and will retain the last data entered. If the RAM EN is enabled, new data will be written into the RAM file (word) defined by the B address field when the clock input is LOW.

There are eight source operand pairs available to the ALU as selected by the I_0 , I_1 , and I_2 instruction inputs. The ALU can

perform eight functions—five logic and three arithmetic. The I_3 , I_4 , and I_5 instruction inputs control this function selection. The carry input, C_n , also affects the ALU results when in the arithmetic mode. The C_n input has no effect in the logic mode. When I_0 through I_5 and C_n are viewed together, the matrix of Table 4 results. This matrix fully defines the ALU/source operand function for each state.

The ALU functions can also be examined on a “task” basis, i.e., add, subtract, AND, OR, etc. In the arithmetic mode, the carry will affect the function performed; while in the logic mode, the carry will have no bearing on the ALU output. Table 5 defines the various logic operations that the Am2901 can perform, and Table 6 shows the arithmetic functions of the device. Both carry-in LOW ($C_n = 0$) and carry-in HIGH ($C_n = 1$) are defined in these operations.

Logic Functions for G, P, C_{n+4} , and OVR

The four signals, G, P, C_{n+4} , and OVR are designed to indicate carry and overflow conditions when the Am2901 is in the add or subtract mode. Table 7 indicates the logic equations for these four signals for each of the eight ALU functions. The R and S inputs are the two inputs selected according to Table 1.

Table 4 Source Operand and ALU Function Matrix

Octal $I_{2,1,0}$ ALU source									
Octal $I_{5,4,3}$	ALU function	0 A, Q	1 A, B	2 O, Q	3 O, B	4 O, A	5 D, A	6 D, Q	7 D, O
0	$C_n = L$ R plus S $C_n = H$	A + Q	A + B	Q	B	A	D + A	D + Q	D
		A + Q + 1	A + B + 1	Q + 1	B + 1	A + 1	D + A + 1	D + Q + 1	D + 1
1	$C_n = L$ S minus R $C_n = H$	Q - A - 1	B - A - 1	Q - 1	B - 1	A - 1	A - D - 1	Q - D - 1	-D - 1
		Q - A	B - A	Q	B	A	A - D	Q - D	-D
2	$C_n = L$ R minus S $C_n = H$	A - Q - 1	A - B - 1	-Q - 1	-B - 1	-A - 1	D - A - 1	D - Q - 1	D - 1
		A - Q	A - B	-Q	-B	-A	D - A	D - Q	D
3	R OR S	$A \vee Q$	$A \vee B$	Q	B	A	$D \vee A$	$D \vee Q$	D
4	R AND S	$A \wedge Q$	$A \wedge B$	0	0	0	$D \wedge A$	$D \wedge Q$	0
5	\bar{R} AND S	$\bar{A} \wedge Q$	$\bar{A} \wedge B$	Q	B	A	$\bar{D} \wedge A$	$\bar{D} \wedge Q$	0
6	R EX-OR S	$A \nabla Q$	$A \nabla B$	Q	B	A	$D \nabla A$	$D \nabla Q$	D
7	R EX-NOR S	$\overline{A \nabla Q}$	$\overline{A \nabla B}$	\bar{Q}	\bar{B}	\bar{A}	$\overline{D \nabla A}$	$\overline{D \nabla Q}$	\bar{D}

+ = Plus; - = Minus; \vee = OR; \wedge = AND; ∇ = EX-OR

Table 5 ALU Logic Mode Functions (C_n Irrelevant)

Octal $I_{5,4,3} I_{2,1,0}$		Group	Function
4	0	AND	$A \wedge Q$
4	1		$A \wedge B$
4	5		$D \wedge A$
4	6		$D \wedge Q$
3	0	OR	$A \vee Q$
3	1		$A \vee B$
3	5		$D \vee A$
3	6		$D \vee Q$
6	0	EX-OR	$A \nabla Q$
6	1		$A \nabla B$
6	5		$D \nabla A$
6	6		$D \nabla Q$
7	0	EX-NOR	$\overline{A \nabla Q}$
7	1		$\overline{A \nabla B}$
7	5		$\overline{D \nabla A}$
7	6		$\overline{D \nabla Q}$
7	2	INVERT	\overline{Q}
7	3		\overline{B}
7	4		\overline{A}
7	7		\overline{D}
6	2	PASS	Q
6	3		B
6	4		A
6	7		D
3	2	PASS	Q
3	3		B
3	4		A
3	7		D
4	2	"ZERO"	0
4	3		0
4	4		0
4	7		0
5	0	MASK	$\overline{A} \wedge Q$
5	1		$\overline{A} \wedge B$
5	5		$\overline{D} \wedge A$
5	6		$\overline{D} \wedge Q$

Pin Definitions

- A_{0-3} The four address inputs to the register stack used to select one register whose contents are displayed through the A port.
- B_{0-3} The four address inputs to the register stack used to select one register whose contents are displayed

through the B port and into which new data can be written when the clock goes LOW.

I_{0-8} The nine instruction control lines to the Am2901, used to determine what data sources will be applied to the ALU ($I_{0,1,2}$), what functions the ALU will perform ($I_{3,4,5}$), and what data is to be deposited in the Q register or the register stack ($I_{6,7,8}$).

O_3 , RAM_3 A shift line at the MSB of the Q register (Q_3) and the register stack (RAM_3). Electrically these lines are three-state outputs connected to TTL inputs internal to the Am2901. When the destination code on $I_{6,7,8}$ indicates an up shift (octal 6 or 7), the three-state outputs are enabled and the MSB of the Q register is available on the Q_3 pin and the MSB of the ALU output is available on the RAM_3 pin. Otherwise, the three-state outputs are OFF (high-impedance) and the pins are electrically LS-TTL inputs. When the destination code calls for a down shift, the pins are used as the data inputs to the MSB of the Q register (octal 4) and RAM (octal 4 or 5).

O_0 , RAM_0 Shift lines like Q_3 and RAM_3 , but at the LSB of the Q register and RAM. These pins are tied to the Q_3 and RAM_3 pins of the adjacent device to transfer data between devices for up and down shifts of the Q register and ALU data.

D_{0-3} Direct data inputs. A 4-bit data field which may be selected as one of the ALU data sources for entering data into the Am2901. D_0 is the LSB.

Y_{0-3} The four data outputs of the Am2901. These are three-state output lines. When enabled, they display either the four outputs of the ALU or the data on the A port of the register stack, as determined by the destination code $I_{6,7,8}$.

\overline{OE} Output enable. When \overline{OE} is HIGH, the Y outputs are OFF; when \overline{OE} is LOW, the Y outputs are active (HIGH or LOW).

\overline{P} , \overline{G} The carry generate and propagate outputs of the Am2901's ALU. These signals are used with the Am2902 for carry-lookahead. See Table 7 for the logic equations.

OVR Overflow. This pin is logically the Exclusive-OR of the carry-in and carry-out of the MSB of the ALU. At the most significant end of the word, this pin indicates that the result of an arithmetic 2's complement operation has overflowed into the sign bit. See Table 7 for logic equation.

$F = 0$ This is an open-collector output which goes HIGH (OFF) if the four ALU outputs F_{0-3} are all LOW. In positive logic, it indicates the result of an ALU operation is 0.

C_n The carry-in to the Am2901's ALU.

C_{n+4} The carry-out of the Am2901's ALU. See Table 7 for equations.

CP The clock to the Am2901. The Q register and register stack outputs change on the clock LOW-to-

Table 6 ALU Arithmetic Mode Functions

Octal $I_{5,4,3} I_{2,1,0}$	$C_n = 0$ (LOW)		$C_n = 1$ (HIGH)	
	Group	Function	Group	Function
0 0	ADD	$A + Q$	ADD plus one	$A + Q + 1$
0 1		$A + B$		$A + B + 1$
0 5		$D + A$		$D + A + 1$
0 6		$D + Q$		$D + Q + 1$
0 2	PASS	Q	Increment	$Q + 1$
0 3		B		$B + 1$
0 4		A		$A + 1$
0 7		D		$D + 1$
1 2	Decrement	$Q - 1$	PASS	Q
1 3		$B - 1$		B
1 4		$A - 1$		A
2 7		$D - 1$		D
2 2	1's complement	$-Q - 1$	2's complement (negate)	$-Q$
2 3		$-B - 1$		$-B$
2 4		$-A - 1$		$-A$
1 7		$-D - 1$		$-D$
1 0	Subtract (1's complement)	$Q - A - 1$	Subtract (2's complement)	$Q - A$
1 1		$B - A - 1$		$B - A$
1 5		$A - D - 1$		$A - D$
1 6		$Q - D - 1$		$Q - D$
2 0		$A - Q - 1$		$A - Q$
2 1		$A - B - 1$		$A - B$
2 5		$D - A - 1$		$D - A$
2 6		$D - Q - 1$		$D - Q$

Table 7

Definitions (+ = OR)

$$\begin{aligned}
 P_0 &= R_0 + S_0 & G_0 &= R_0 S_0 & C_4 &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_n \\
 P_1 &= R_1 + S_1 & G_1 &= R_1 S_1 & C_3 &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_n \\
 P_2 &= R_2 + S_2 & G_2 &= R_2 S_2 & & \\
 P_3 &= R_3 + S_3 & G_3 &= R_3 S_3 & &
 \end{aligned}$$

$I_{5,4,3}$	Function	\overline{P}	\overline{G}	C_{n+4}	OVR
0	R + S	$\overline{P_3 P_2 P_1 P_0}$	$\overline{G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0}$	C_4	$C_3 \vee C_4$
1	S − R	← Same as R + S equations, but substitute $\overline{R_i}$ for R_i in definitions →			
2	R − S	← Same as R + S equations, but substitute $\overline{S_i}$ for S_i in definitions →			
3	R ∨ S	LOW	$P_3 P_2 P_1 P_0$	$\overline{P_3 P_2 P_1 P_0} + C_n$	$\overline{P_3 P_2 P_1 P_0} + C_n$
4	R ∧ S	LOW	$\overline{G_3 + G_2 + G_1 + G_0}$	$G_3 + G_2 + G_1 + G_0 + C_n$	$G_3 + G_2 + G_1 + G_0 + C_n$
5	$\overline{R} \wedge S$	LOW	← Same as R ∧ S equations, but substitute $\overline{R_i}$ for R_i in definitions →		
6	R ∇ S	← Same as $\overline{R} \vee S$, but substitute $\overline{R_i}$ for R_i in definitions →			
7	$\overline{\overline{R} \nabla \overline{S}}$	$G_3 + G_2 + G_1 + G_0$	$G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$	$\frac{G_3 + P_3 G_2 + P_3 P_2 G_1}{+ P_3 P_2 P_1 P_0 (G_0 + \overline{C_n})}$	See note

Note: $[\bar{P}_2 + \bar{G}_2 \bar{P}_1 + \bar{G}_2 \bar{G}_1 \bar{P}_0 + \bar{G}_2 \bar{G}_1 \bar{G}_0 C_n] \nabla [\bar{P}_3 + \bar{G}_3 \bar{P}_2 + \bar{G}_3 \bar{G}_2 \bar{P}_1 + \bar{G}_3 \bar{G}_2 \bar{G}_1 \bar{P}_0 + \bar{G}_3 \bar{G}_2 \bar{G}_1 \bar{G}_0 C_n]$

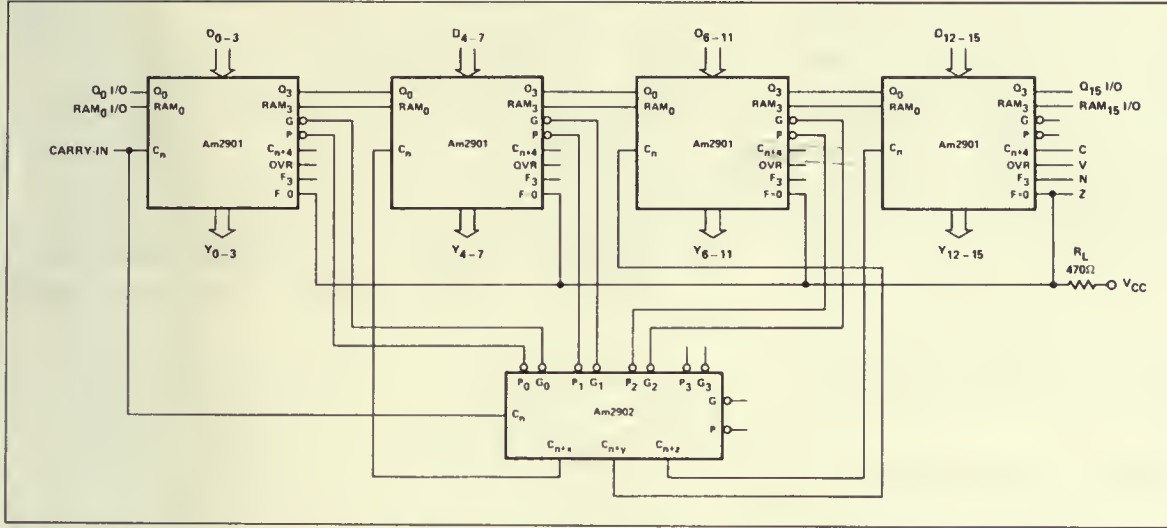


Fig. 5. Four Am2901's in a 16-bit CPU using the Am2902 for carry lookahead.

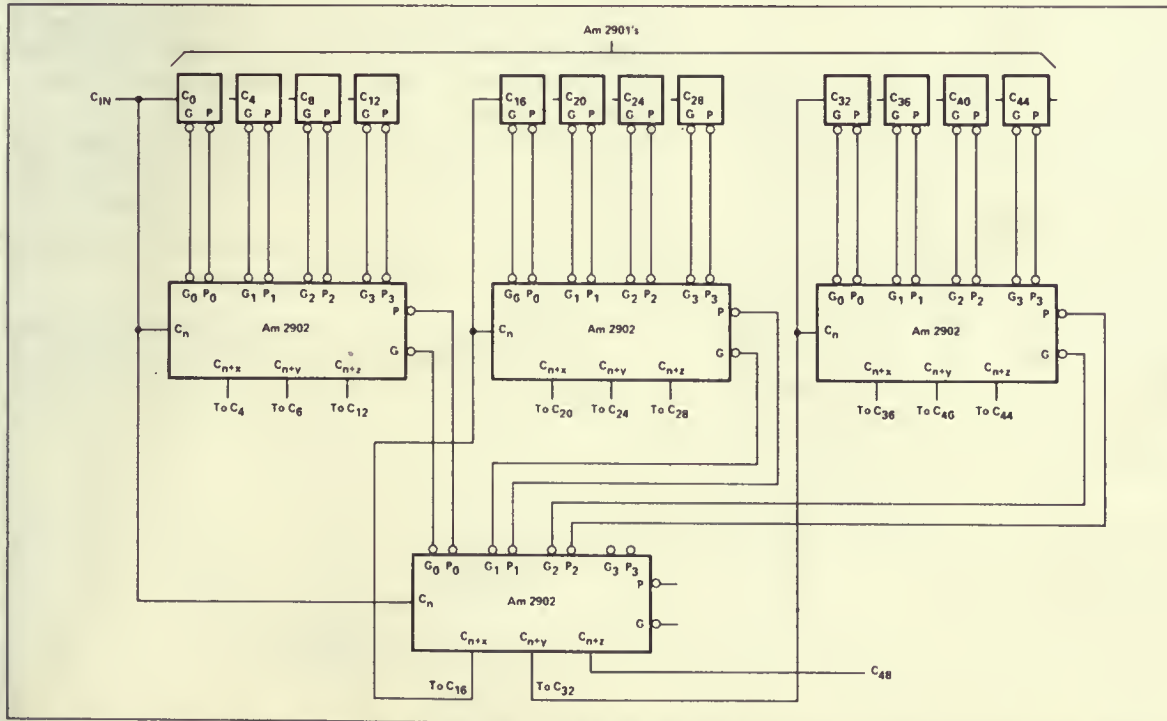


Fig. 6. Carry lookahead scheme for 48-bit CPU using 12 Am2901's. The carry-out flag (C48) should be taken from the lower Am2902 rather than the right-most Am2901 for higher speed.

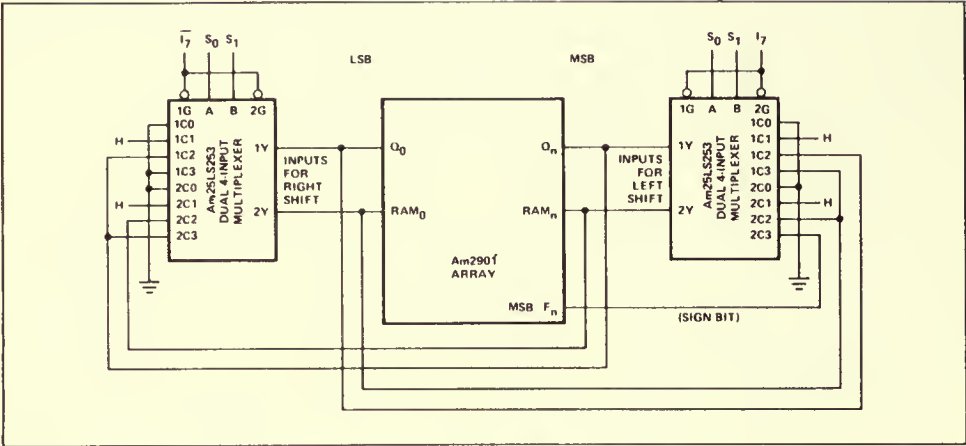


Fig. 7. Three-state multiplexers-used on shift I/O lines.

Arithmetic A double-length arithmetic shift if Q is also shifted. On an up shift a zero is loaded into the Q -register LSB and the Q -register MSB is loaded into the RAM LSB. On a down shift, the RAM LSB is loaded into the Q -register MSB and the ALU output MSB (F_n , the sign bit) is loaded into the RAM MSB. (This same bit will also be in the next less significant RAM bit.)

Hardware Multiplication

Figure 8 illustrates the interconnections for a hardware multiplication using the Am2901. The system shown uses two devices for 8×8 multiplication, but the expansion to more bits is simple—the significant connections are at the LSB and MSB only.

The basic technique used is the “add and shift” algorithm. One clock cycle is required for each bit of the multiplier. On each cycle, the LSB of the multiplier is examined; if it is a 1, then the multiplicand is added to the partial product to generate a new partial product. The partial product is then shifted one place toward the LSB, and the multiplier is also shifted one place toward the LSB. The old LSB of the multiplier is discarded. The

cycle is then repeated on the new LSB of the multiplier available at Q_0 .

The multiplier is in the Am2901 Q register. The multiplicand is in one of the registers in the register stack, R_a . The product will be developed in another of the registers in the stack, R_b .

The A address inputs are used to address the multiplicand in R_a , and the B address inputs are used to address the partial product in R_b . On each cycle, R_a is conditionally added to R_b , depending on the LSB of Q as read from the Q_0 output, and both the Q and the ALU output are shifted left one place. The instruction lines to the Am2901 on every cycle will be:

- $I_{8,7,6} = 4$ (shift register stack input and Q register left)
- $I_{5,4,3} = 0$ (Add)
- $I_{2,1,0} = 1$ or 3 (select A, B or O, B as ALU sources)

Figure 8 shows the connections for multiplication. The circled numbers refer to the paragraphs below.

- 1 The adjacent pins of the Q register and RAM shifters are connected together so that the Q registers of both (or all) Am2901's shift left or right as a unit. Similarly, the entire

Table 8

Code			Source of new data				Shift	Type
I_7	S_1	S_0	Q_0	Q_n	RAM_0	RAM_n		
H	L	L	0	Q_{n-1}	0	F_{n-1}	Up (Right)	Zero
H	L	H	1	Q_{n-1}	1	F_{n-1}		One
H	H	L	Q_n	Q_{n-1}	F_n	F_{n-1}		Rotate
H	H	H	0	Q_{n-1}	Q_n	F_{n-1}		Arithmetic
L	L	L	Q_1	0	F_1	0	Down (Left)	Zero
L	L	H	Q_1	1	F_1	1		One
L	H	L	Q_1	Q_0	F_1	F_0		Rotate
L	H	H	Q_1	F_0	F_1	$RAM_n = RAM_{n-1} = F_n$		Arithmetic

push/pop stack; or (4) a program counter register (which usually contains the last address plus one). The push/pop stack includes certain control lines so that it can efficiently execute nested subroutine linkages. Each of the four outputs can be ORed with an external input for conditional skip or branch instructions, and a separate line forces the outputs to all zeros. The outputs are three-state.

Architecture of the Am2909

A detailed logic diagram is shown in Fig. 11. The device contains a four-input multiplexer that is used to select either the address register, direct inputs, microprogram counter, or file as the source of the next microinstruction address. This multiplexer is controlled by the S_0 and S_1 inputs.

The address register consists of four D -type, edge-triggered flip-flops with a common clock enable. When the address register enable is LOW, new data is entered into the register on the clock LOW-to-HIGH transition. The address register is available at the multiplexer as a source for the next microinstruction address. The direct input is a 4-bit field of inputs to the multiplexer and can be selected as the next microinstruction address.

The Am2909 contains a microprogram counter (μPC) that is composed of a 4-bit incrementer followed by a 4-bit register. The

incrementer has carry-in (C_n) and carry-out (C_{n+4}) such that cascading to larger word lengths is straightforward. The μPC can be used in either of two ways. When the least significant carry-in to the increment is HIGH, the microprogram register is loaded on the next clock cycle with the current Y output word plus one ($Y + 1 \rightarrow \mu PC$). Thus sequential microinstructions can be executed. If this least significant C_0 is LOW, the incrementer passes the Y output word unmodified and the microprogram register is loaded with the same Y word on the next cycle ($Y \rightarrow \mu PC$). Thus, the same microinstruction can be executed any number of times by using the least significant C_n as the control.

The last source available at the multiplexer input is the 4×4 file (stack). The file is used to provide return address linkage when executing microsubroutines. The file contains a built-in stack pointer (SP) which always points to the last file word written. This allows stack reference operations (looping) to be performed without a push or pop.

The stack pointer operates as an up/down counter with separate push/pop and file enable inputs. When the file enable input is LOW and the push/pop input is HIGH, the PUSH operation is enabled. This causes the stack pointer to increment and the file to be written with the required return linkage—the next microinstruction address following the subroutine jump which initiated the PUSH.

If the file enable input is LOW and the push/pop control is LOW, a POP operation occurs. This implies the usage of the return linkage during this cycle and thus a return from subroutine. The next LOW-to-HIGH clock transition causes the stack pointer to decrement. If the file enable is HIGH, no action is taken by the stack pointer regardless of any other input.

The stack pointer linkage is such that any combination of pushes, pops, and stack references can be achieved. One microinstruction subroutines can be performed. Since the stack is 4 words deep, up to four microsubroutines can be nested.

The ZERO input is used to force the four outputs to the binary zero state. When the ZERO input is LOW, all Y outputs are LOW regardless of any other inputs (except \overline{OE}). Each Y output bit also has a separate OR input such that a conditional logic 1 can be forced at each Y output. This allows jumping to different microinstructions on programmed conditions.

The Am2909 features three-state Y outputs. These can be particularly useful in military designs requiring external ground support equipment (GSE) to provide automatic checkout of the microprocessor. The internal control can be placed in the high-impedance state, and preprogrammed sequences of microinstructions can be executed via external access to the control ROM/PROM.

Definition of Terms

A set of symbols is used to represent various internal and external registers and signals used with the Am2909. Since its principal

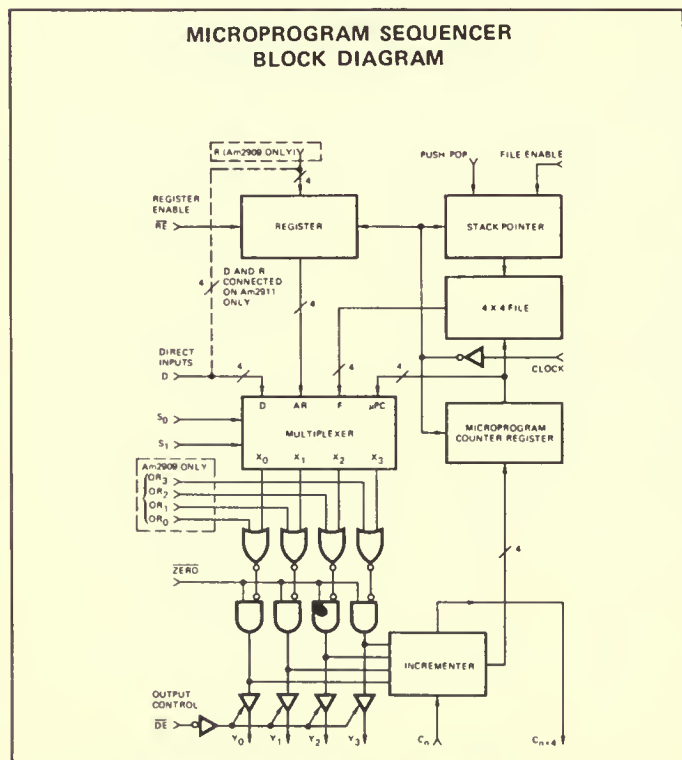


Fig. 10. Microprogram sequencer block diagram.



181

application is as a controller for a microprogram store, it is necessary to define some signals associated with the microcode itself. Figure 12 illustrates the basic interconnection of Am2909, memory, and microinstruction register. The definitions here apply to this architecture.

Inputs to Am2909

S_1, S_0	Control lines for address source selection.
\overline{FE}, PUP	Control lines for push/pop stack.
\overline{RE}	Enable line for internal address register.
OR_i	Logic OR inputs on each address output line.
\overline{ZERO}	Logic AND input on the output lines.
\overline{OE}	Output enable. When \overline{OE} is HIGH, the Y outputs are OFF (high impedance).
C_n	Carry-in to the incrementer.
R_i	Inputs to the internal address register.
D_i	Direct inputs to the multiplexer.
CP	Clock input to the AR and μPC register and push-pop stack.

Outputs from the Am2909

Y_i	Address outputs from Am2909 (address inputs to control memory).
C_{n+4}	Carry-out from the incrementer.

Internal Signals

μPC	Contents of the microprogram counter.
REG	Contents of the register.
STK0-STK3	Contents of the push/pop stack. By definition, the word in the 4×4 file addressed by the stack pointer is STK0. Conceptually data is pushed into the stack at STK0; a subsequent push moves STK0 to STK1; a pop implies $STK3 \rightarrow STK2 \rightarrow STK1 \rightarrow STK0$. Physically, only the stack pointer changes when a push or pop is performed. The data does not move. I/O occurs at STK0.
SP	Contents of the stack pointer.

External to the Am2909

A	Address to the control memory.
I(A)	Instruction in control memory at address A.
μWR	Contents of the microword register (at output of control memory). The microword register contains the instruction currently being executed.
T_n	Time period (cycle) n.

Operation of the Am2909

Figure 13 lists the select codes for the multiplexer. The two bits applied from the microword register (and additional combination logic for branching) determine which data source contains the

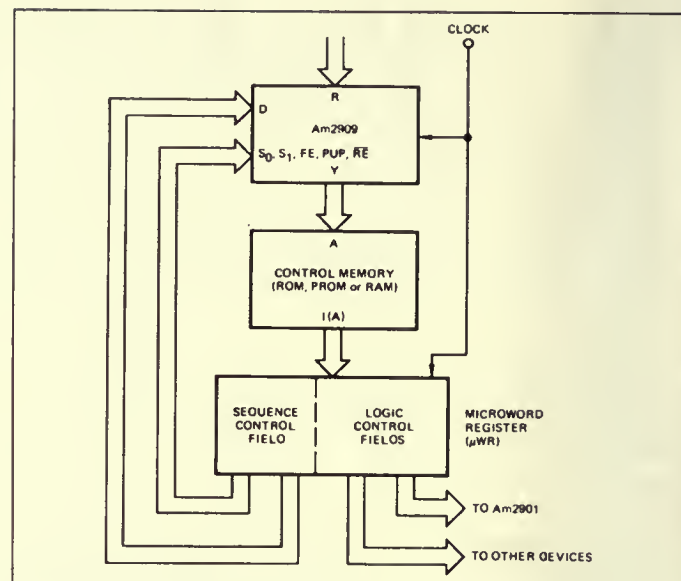


Fig. 12. Microprogram sequencer control.

address for the next microinstruction. The contents of the selected source will appear on the Y outputs. Figure 13 also shows the truth table for the output control and for the control of the push/pop stack. Table 9 shows in detail the effect of S_0 , S_1 , \overline{FE} , and PUP on the Am2909. These four signals define what address appears on the Y outputs and what the state of all the internal registers will be following the clock LOW-to-HIGH edge. In this illustration, the microprogram counter is assumed to contain initially some word J, the address register some word K, and the four words in the push/pop stack R_a through R_d .

Figure 14 illustrates the execution of a subroutine using the Am2909. The configuration of Fig. 11 is assumed. The instruction being executed at any given time is the one contained in the microword register (μWR). The contents of the μWR also control (indirectly, perhaps) the four signals S_0 , S_1 , \overline{FE} , and PUP. The starting address of the subroutine is applied to the D inputs of the Am2909 at the appropriate time.

In the columns on the left is the sequence of microinstructions to be executed. At address $J + 2$, the sequence control portion of the microinstruction contains the command "Jump to subroutine at A." At the time T_2 , this is in the μWR , and the Am2909 inputs are set up to execute the jump and save the return address. The subroutine address A is applied to the D inputs from the μWR and appears on the Y outputs. The first instruction of the subroutine, I(A), is accessed and is at the inputs of the μWR . On the next clock transition, I(A) is loaded into the μWR for execution, and the return address $J + 3$ is pushed onto the stack. The return instruction is executed at T_5 .

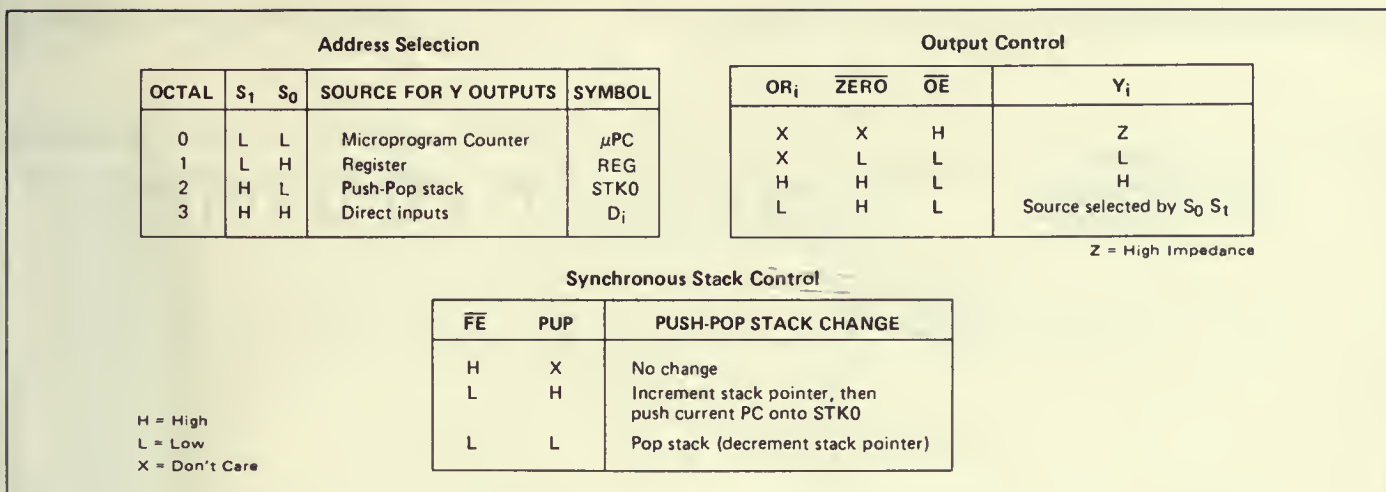


Fig. 13

Table 9 Output and Internal Next-Cycle Register States for Am2909

Cycle	S ₁ S ₀ FE, PUP	μPC	REG	STK0	STK1	STK2	STK3	Y _{OUT}	Comment	Principal use
N N + 1	0 0 0 0	J J + 1	K K	Ra Rb	Rb Rc	Rc Rd	Rd Ra	J ...	Pop stack	End loop
N N + 1	0 0 0 1	J J + 1	K K	Ra J	Rb Ra	Rc Rb	Rd Rc	J ...	Push μPC	Set up loop
N N + 1	0 0 1 X	J J + 1	K K	Ra Ra	Rb Rb	Rc Rc	Rd Rd	J ...	Continue	Continue
N N + 1	0 1 0 0	J K + 1	K K	Ra Rb	Rb Rc	Rc Rd	Rd Ra	K ...	Pop stack; Use AR for address	End loop
N N + 1	0 1 0 1	J K + 1	K K	Ra J	Rb Ra	Rc Rb	Rd Rc	K ...	Push μPC; Jump to address in AR	JSR AR
N N + 1	0 1 1 X	J K + 1	K K	Ra Ra	Rb Rb	Rc Rc	Rd Rd	K ...	Jump to address in AR	JMP AR
N N + 1	1 0 0 0	J Ra + 1	K K	Ra Rb	Rb Rc	Rc Rd	Rd Ra	Ra ...	Jump to address in STK0; Pop stack	RTS
N N + 1	1 0 0 1	J Ra + 1	K K	Ra J	Rb Ra	Rc Rb	Rd Rc	Ra ...	Jump to address in STK0; Push μPC	
N N + 1	1 0 1 X	J Ra + 1	K K	Ra Ra	Rb Rb	Rc Rc	Rd Rd	Ra ...	Jump to address in STK0	Stack ref (loop)
N N + 1	1 1 0 0	J D + 1	K K	Ra Rb	Rb Rc	Rc Rd	Rd Ra	D ...	Pop stack; Jump to address on D	End loop
N N + 1	1 1 0 1	J D + 1	K K	Ra J	Rb Ra	Rc Rb	Rd Rc	D ...	Jump to address on D; Push μPC	JSR D
N N + 1	1 1 1 X	J D + 1	K K	Ra Ra	Rb Rb	Rc Rc	Rd Rd	D ...	Jump to address on D	JMP D

X = Don't care, 0 = LOW, 1 = HIGH, Assume C_n = HIGH

Note: STK0 is the location addressed by the stack pointer.

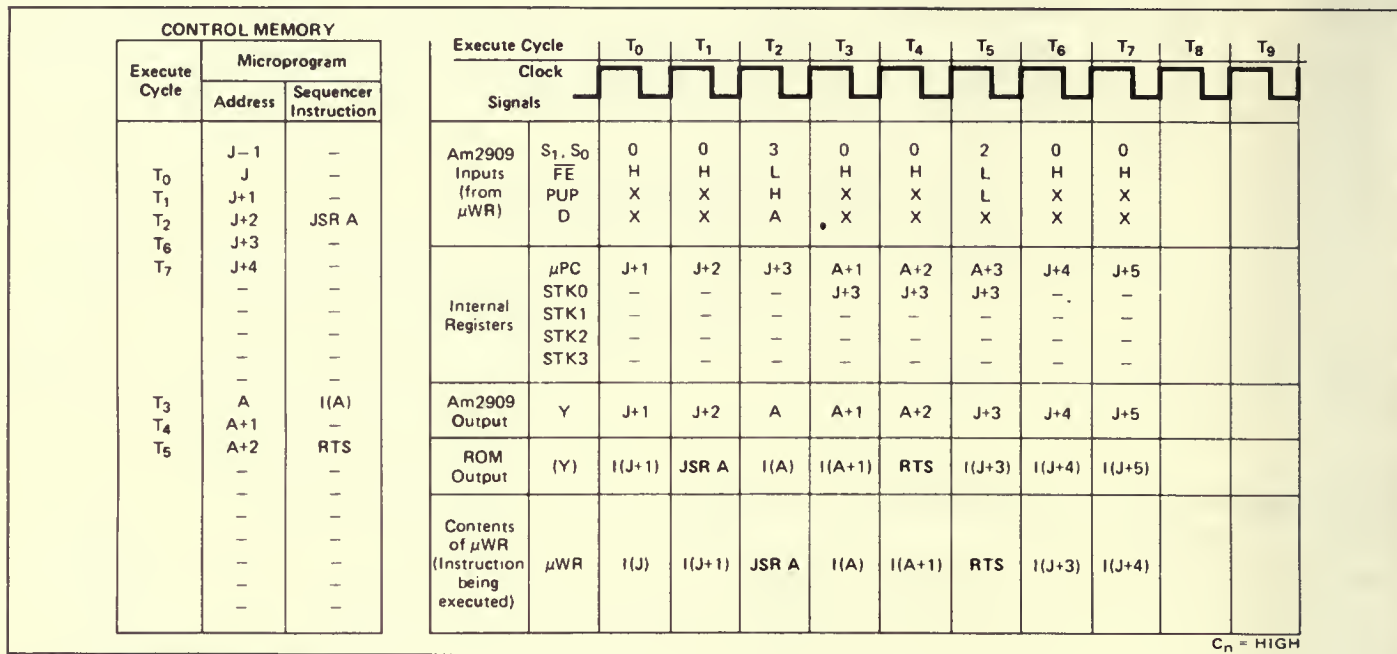


Fig. 14. Subroutine execution.

APPENDIX 1 AM2909 ISP DESCRIPTION

```

AM2909 :=
begin
    ! ISPS description of AM0 AM2909 bit slice microprogram sequencer.

    ! The AM2909 is a 4 bit slice (expandable to 4*n bits) address controller.
    ! The controller is designed to be used with the AM2901 microprocessor
    ! slice and external memory.

    ! Simulation of the AM2909 alone is possible, but emulation of any
    ! computer systems requires that this description be joined with
    ! the AM2901 description.

    **PC.State**
    uPC<3:0> := incr<3:0>.      ! Microprogram counter
    REG<3:0>.  ! Address register
    SP<1:0>.   ! Stack pointer
    STACK<0:3><3:0>. ! Stack register file

    **External.State**
    Cn<>.      ! Carry in
    Cn4<> := incr<4>.      ! Carry out
    D<3:0>.   ! Direct inputs
    FE<>.     ! Stack register file enable
    DE<>.     ! Output enable control line
    OR.<3:0>. ! Logical OR inputs
    PUP<>.    ! Push/pop control line
    RC<3:0>.  ! Address register inputs
    RE<>.     ! Register enable control line
    SK<1:0>.  ! Address select control lines
    ZERD<>.   ! Zero out control line

    **Implementation.Variables**
    incr<4:0>.      ! Incrementor
    macro z := ['1111], ! High impedance constant

    **Operation.Cycle**(us)
    start(main) := ! Initialization
    begin
        OE = FE = 0;
        RE = ZERD = 1 next

        run := ! Basic operation loop
        begin
            Y() next ! Put out selected address
            IF not FE => ! Perform any stack operation
                begin
                    DECODE PUP =>
                    begin
                        '0\pop := {SP = SP - 1},
                        '1\push := {SP = SP + 1 next
                        STACK[SP] = uPC
                    end
                end next
            IF not RE => REG = R; ! Load register if enabled
            incr = uPC + Cn next ! Increment pc
            RESTART run
        end

        **Address.Source.Selection**(us)
        Y{<3:0> :=
        begin
            DECODE ZERO @ OE =>
            begin
                '00 := Y = uPC = '0000,
                ['01..'11] := Y = z,
                '10 := DECODE S =>
                begin
                    '00 := Y = uPC = uPC or DR.,
                    '01 := Y = uPC = REG or DR.,
                    '10 := Y = uPC = STACK[SP] or DR.,
                    '11 := Y = uPC = 0 or DR.
                end
            end
        end

    end
end ! End of AM2909 description

```


APPENDIX 2 AM2901 ISP DESCRIPTION

```

AM2901 :=
begin
! ISPS description of the AMD 2901 4 bit slice microprocessor.
! Page 1 contains the declaration of all actual and implementation
! variables.
! Page 2 describes the basic instruction cycle and the source and
! destination access computations.
! Page 3 defines the actual instruction execution process.
! Page 4 contains routines that aid in computation of the carry generate
! (G), overflow (DVR), and carry propagate (P).
! outputs.

**PC.State**
R<3:0>,      ! R inputs to ALU
S<3:0>,      ! S inputs to ALU
F<3:0> := ALU<3:0>, ! Output from ALU
Q<3:0>,      ! Output from Q register

**MP.State**
RAM[0:15]<3:0>, ! 16 x 4 bit 2 port RAM
A.LATCH<3:0>,  ! A RAM port latch
B.LATCH<3:0>,  ! B RAM port latch

**External.State**
A<3:0>,      ! A RAM port input address
B<3:0>,      ! B RAM port input address
D<3:0>,      ! Direct data inputs
Y<3:0>,      ! Data outputs
OE<3:0>,      ! Output enable (tristate control)
PC<3:0>,      ! Carry propagate
GC<3:0>,      ! Carry generate
DVR<3:0>,      ! Overflow
FEQLD<3:0>,   ! High if ALU output = 0
Cn<3:0>,      ! Carry in
Cn4<3:0> := ALU<4>, ! Carry out
RAM0<3:0>,    ! Low order shift input/output
RAM3<3:0>,    ! High order shift input/output
Q0<3:0>,      ! Low order Q shift input/output
Q3<3:0>,      ! High order Q shift input/output

**Instruction.Format**
I<8:0>,      ! Instruction inputs
src<2:0> := I<2:0>, ! Source operand field
op<2:0> := I<5:3>, ! Operation field
dest<2:0> := I<8:8>, ! Destination operand field

**Implementation.Variables**
ALU<4:0>,      ! ALU + carry output
ctemp<3:0>,    ! Temporary for generating carry
macro Z := '1111', ! Tristate constant

**Instruction.Cycle**(us)
start(main) := ! Initialization
begin
DE := DVR = FEQLD = Cn4 = 0;
P = G = 1 next

run := ! Main instruction cycle
begin
source() next
exec() next
destination() next
RESTART run
end
end.

**Access.Computation**(us)
source := ! Source calculation
begin
A.LATCH = RAM[A];
B.LATCH = RAM[B] next
DECODE src =>
begin
#0 := (R = A.LATCH; S = Q),
#1 := (R = A.LATCH; S = B.LATCH),
#2 := (R = 0; S = Q),
#3 := (R = 0; S = B.LATCH),
#4 := (R = 0; S = A.LATCH),
#5 := (R = 0; S = A.LATCH),
#6 := (R = 0; S = Q),
#7 := (R = 0; S = 0)
end
end.

destination := ! Destination calculation
begin
DECODE dest =>
begin
#0 := (Q = F; Y = F),
#1 := (Y = F),
#2 := (Y = RAM[A]; RAM[B] = F),
#3 := (Y = F; RAM[B] = F),
#4 := (Y = F; RAM[B] @ RAM0 = RAM3 @ F; Q @ Q0 = Q3 @ Q),
#5 := (Y = F; RAM[B] @ RAM0 = RAM3 @ F),
#6 := (Y = F; RAM3 @ RAM[B] = F @ RAM0; Q3 @ Q = Q @ Q0),
#7 := (Y = F; RAM3 @ RAM[B] = F @ RAM0)
end
end.

**Instruction.Execution**(us)
exec :=
begin
DECODE op =>
begin
#0 := (ALU = R + S; ! R + S
P = ((R or S) neq(us) '1111);
G = g.compute(R, S);
DVR = ALU<4> xor ALU<3>),
#1 := (ALU = S - R; ! S - R
P = (((not R) or S) neq(us) '1111);
G = g.compute((not R), S);
DVR = ALU<4> xor ALU<3>),
#2 := (ALU = R - S; ! R - S
P = (R or (not S) neq(us) '1111);
G = g.compute(R, (not S));
DVR = ALU<4> xor ALU<3>),
#3 := (ALU = R or S; ! R or S
P = 0;
G = ((R or S) eq1(us) '1111) next
Cn4 = DVR = (not G) or Cn),
#4 := (ALU = R and S; ! R and S
P = 0;
G = ((R and S) eq1(us) '0000) next
Cn4 = DVR = (not G) or Cn),
#5 := (ALU = ((not R) and S); ! (not R) and S
P = 0;
G = (((not R) and S) eq1(us) '0000) next
Cn4 = DVR = (not G) or Cn),
#6 := (ALU = R xor S; ! R xor S
P = (((not R) and S) neq(us) '0000);
G = not g.compute((not R), S);
Cn4 = cB7(not R, S);
DVR = ovrB7(not R, S)),
#7 := (ALU = R eqv S; ! R eqv S
P = ((R and S) neq(us) '0000);
G = not g.compute(R, S);
Cn4 = cB7(R, S);
DVR = ovrB7(R, S))
end next
FEQLD = (F eqv '0000)
end.

** Service.Facilities **(us)
g.compute(r.<3:0>, s.<3:0><> :=
begin
g.compute = (((r. end s.) and ('10(r. OR s.)<3:1>)
and ('110(r. or s.)<3:2>)
and ('1110(r. or s.)<3>))
eq1(us) '0000)
end,
cB7(r.<3:0>, s.<3:0><> := ! Carry for DP 6 and 7
begin
cB7 = not(((r. or s.) eq1(us) '1111) and ((r. and s.)<3:1> neq(us) '000)
and ((r. end s.)<0> or (not Cn)))
end,
ovrB7(r.<3:0>, s.<3:0><> := ! Overflow for DP 6 and 7
begin
ctemp = (r. or s.) and (r. and s.) or ('000 @ Cn)
and ('11 @ (r. or s.)<2:1>) and ('111 @ (r. or s.)<2>) next
ovrB7 = (((ctemp<2:0> eq1(us) '000) and ctemp<3>)
or (not((r. or s.)<3> or (ctemp<2:0> eq1 '000))))
end,
end ! End of AM2901 description

```

Chapter 14

The Am2903/2910¹

General Description of the Am2903

The Am2903 is a 4-bit expandable bipolar microprocessor slice. The Am2903 performs all functions performed by the industry standard Am2901A and, in addition, provides a number of significant enhancements that are especially useful in arithmetic-oriented processors. Infinitely expandable memory and three-port, three-address architecture are provided by the Am2903. In addition to its complete arithmetic and logic instruction set, the Am2903 provides a special set of instructions which facilitate the implementation of multiplication, division, normalization, and other previously time-consuming operations. The Am2903 is supplied in a 48-pin dual in-line package.

Architecture of the Am2903

The Am2903 is a high-performance cascadable 4-bit bipolar microprocessor slice designed for use in CPU's, peripheral controllers, microprogrammable machines, and numerous other applications. The 9-bit microinstruction selects the ALU sources, function, and destination. The Am2903 is cascadable with full lookahead or ripple carry, has three-state outputs, and provides various ALU status flag outputs. Advanced low-power Schottky processing is used to fabricate this 48-pin LSI circuit.

All data paths within the device are 4 bits wide. As shown in Fig. 1, the device consists of a 16-word by 4-bit two-port RAM with latches on both output ports, a high-performance ALU and shifter, a multi-purpose Q register with shifter input, and a 9-bit instruction decoder.

Two-Port RAM

Any two RAM words addressed at the A and B address ports can be read simultaneously at the respective RAM A and B output ports. Identical data appears at the two output ports when the same address is applied to both address ports. The latches at the RAM output ports are transparent when the clock input, CP, is HIGH, and they hold the RAM output data when CP is LOW. Under control of the \overline{OE}_B three-state output enable, RAM data can be read directly at the Am2903 DB I/O port.

External data at the Am2903 Y I/O port can be written directly

into the RAM, or ALU shifter output data can be enabled onto the Y I/O port and entered into the RAM. Data is written into the RAM at the B address when the write enable input, \overline{WE} , is LOW and the clock input, CP, is LOW.

Arithmetic Logic Unit

The Am2903 high-performance ALU can perform seven arithmetic and nine logic operations on two 4-bit operands. Multiplexers at the ALU inputs provide the capability to select various pairs of ALU source operands. The \overline{E}_A input selects either the DA external data input or RAM output port A for use as one ALU operand, and the \overline{OE}_B and I_0 inputs select RAM output port B, DB external data input, or the Q-register content for use as the second ALU operand. Also, during some ALU operations, zeros are forced at the ALU operand inputs. Thus, the Am2903 ALU can operate on data from two external sources, from an internal and external source, or from two internal sources. Table 1 shows all possible pairs of ALU source operands as a function of the \overline{E}_A , \overline{OE}_B , and I_0 inputs.

When instruction bits I_4 , I_3 , I_2 , I_1 , and I_0 are LOW, the Am2903 executes special functions. Table 4 defines these special functions and the operation which the ALU performs for each. When the 2903 executes instructions other than the nine special functions, the ALU operation is determined by instruction bits I_4 , I_3 , I_2 , and I_1 . Table 2 defines the ALU operation as a function of these four instruction bits.

Am2903's may be cascaded in either a ripple carry or lookahead carry fashion. When a number of Am2903's are cascaded, each slice must be programmed to be a most significant slice (MSS), intermediate slice (IS), or least significant slice (LSS) of the array. The carry generate, \overline{G} , and carry propagate, \overline{P} , signals required for a lookahead carry scheme are generated by the Am2903 and are available as outputs of the least significant and intermediate slices.

The Am2903 also generates a carry-out signal, C_{n+4} , which is generally available as an output of each slice. Both the carry-in, C_n , and carry-out, C_{n+4} , signals are active HIGH. The ALU generates two other status outputs. These are negative, N, and overflow, OVR. The N output is generally the most significant (sign) bit of the ALU output and can be used to determine positive or negative results. The OVR output indicates that the arithmetic operation being performed exceeds the available 2's complement number range. The N and OVR signals are available as outputs of the most significant slice. Thus the multi-purpose \overline{G} /N and \overline{P} /OVR outputs indicate \overline{G} and \overline{P} at the least significant and intermediate slices, and sign and overflow at the most significant slice. To some extent, the meanings of the C_{n+4} , \overline{P} /OVR, and \overline{G} /N signals vary with the ALU function being performed. Refer to Table 5 for an exact definition of these four signals as a function of the Am2903 instruction.

¹Abstracted from "Am2903, The Superslice" and "Am2910 Microprogram Controller" specification sheets, Advanced Micro Devices, Inc., 1978.

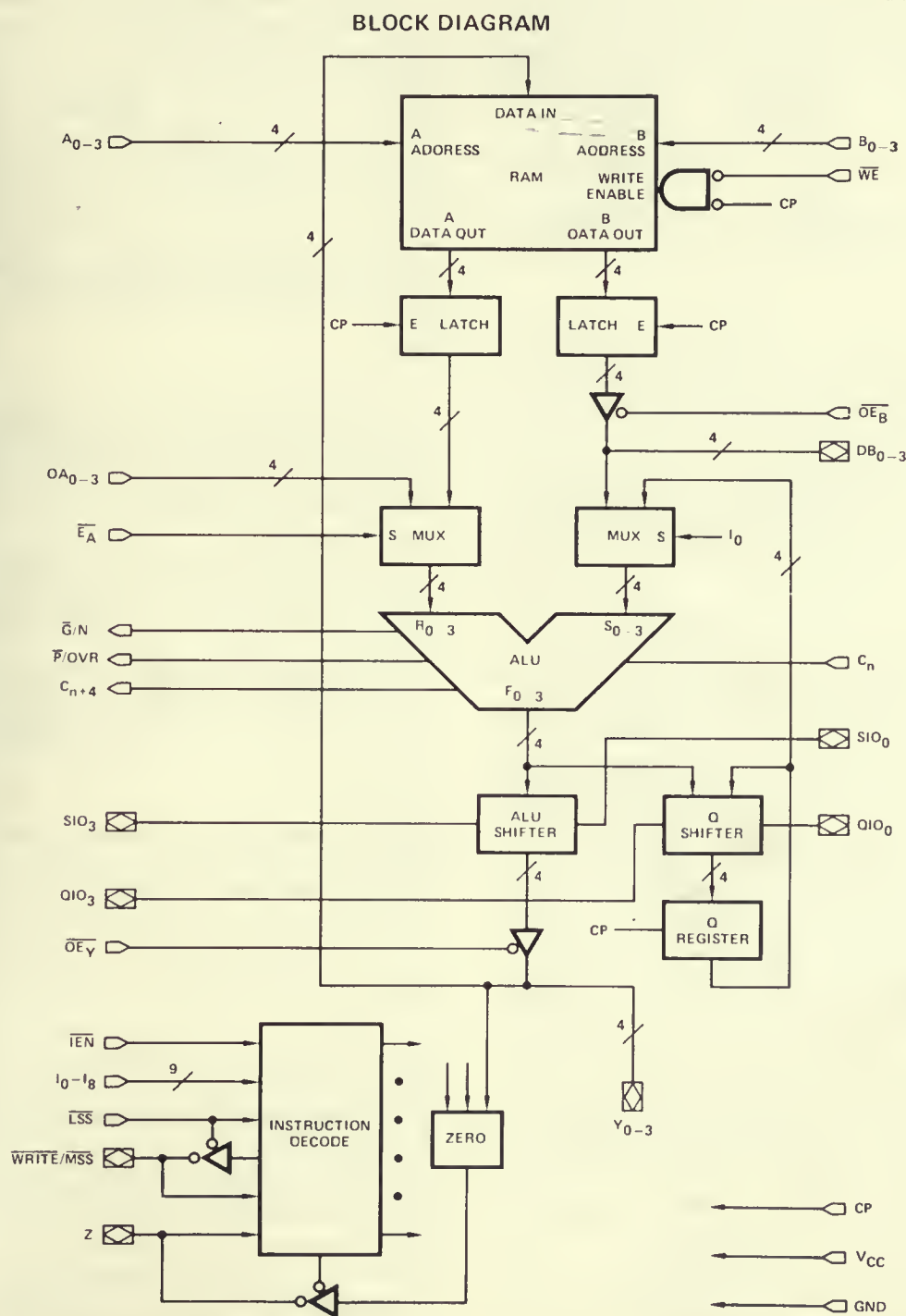


Fig. 1. Block diagram.

Table 1 ALU Operand Sources

\overline{E}_A	I_0	\overline{OE}_B	ALU operand R	ALU operand S
L	L	L	RAM output A	RAM output B
L	L	H	RAM output A	DB ₀₋₃
L	H	X	RAM output A	Q Register
H	L	L	DA ₀₋₃	RAM output B
H	L	H	DA ₀₋₃	DB ₀₋₃
H	H	X	DA ₀₋₃	Q Register

L = LOW H = HIGH X = don't care

ALU Shifter

Under instruction control, the ALU shifter passes the ALU output (F) non-shifted, shifts it up one bit position (2F), or shifts it down one bit position (F/2). Both arithmetic and logical shift operations are possible. An arithmetic shift operation shifts data around the most significant (sign) bit position of the most significant slice, and a logical shift operation shifts data through this bit position (see Fig. 2). SIO₀ and SIO₃ are bidirectional serial shift inputs/outputs. During a shift-up operation, SIO₀ is generally a serial shift input

Table 2 ALU Functions

I_4	I_3	I_2	I_1	Hex code	ALU functions
L	L	L	L	0	$I_0 = L$ Special functions $I_0 = H$ $F_i = \text{HIGH}$
L	L	L	H	1	$F = S \text{ Minus } R \text{ Minus } 1 \text{ Plus } C_n$
L	L	H	L	2	$F = R \text{ Minus } S \text{ Minus } 1 \text{ Plus } C_n$
L	L	H	H	3	$F = R \text{ Plus } S \text{ Plus } C_n$
L	H	L	L	4	$F = S \text{ Plus } C_n$
L	H	L	H	5	$F = \overline{S} \text{ Plus } C_n$
L	H	H	L	6	$F = R \text{ Plus } C_n$
L	H	H	H	7	$F = \overline{R} \text{ Plus } C_n$
H	L	L	L	8	$F_i = \text{LOW}$
H	L	L	H	9	$F_i = \overline{R}_i \text{ AND } S_i$
H	L	H	L	A	$F_i = R_i \text{ Exclusive - NOR } S_i$
H	L	H	H	B	$F_i = R_i \text{ Exclusive - OR } S_i$
H	H	L	L	C	$F_i = R_i \text{ AND } S_i$
H	H	L	H	D	$F_i = R_i \text{ NOR } S_i$
H	H	H	L	E	$F_i = R_i \text{ NAND } S_i$
H	H	H	H	F	$F_i = R_i \text{ OR } S_i$

L = LOW H = HIGH i = 0 to 3

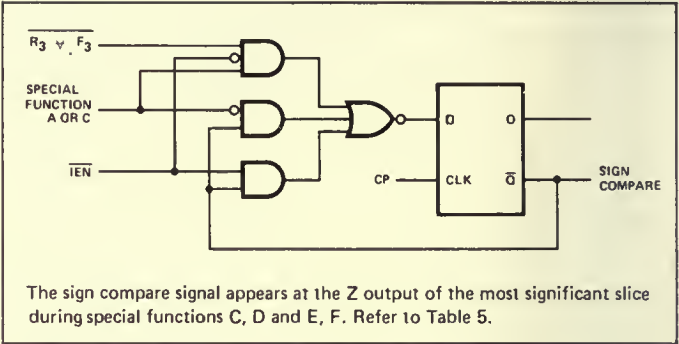


Fig. 2

and SIO₃ a serial shift output. During a shift down operation, SIO₃ is generally a serial shift input and SIO₀ a serial shift output.

To some extent, the meaning of the SIO₀ and SIO₃ signals is instruction-dependent. Refer to Tables 3 and 4 for an exact definition of these pins.

The ALU shifter also provides the capability to sign-extend at slice boundaries. Under instruction control, the SIO₀ (sign) input can be extended through Y₀, Y₁, Y₂, and Y₃ and propagated to the SIO₃ output.

A cascadable 5-bit parity generator/checker is designed into the Am2903 ALU shifter and provides ALU error detection capability. Parity for the F₀, F₁, F₂, and F₃ ALU outputs and SIO₃ input is generated and, under instruction control, is made available at the SIO₀ output.

The instruction inputs determine the ALU shifter operation. Table 4 defines the special functions and the operation the ALU shifter performs for each. When the Am2903 executes instructions other than the nine special functions, the ALU shifter operation is determined by instruction bits I₈I₇I₆I₅. Table 3 defines the ALU shifter operation as a function of these four bits.

Q Register

The Q register is an auxiliary 4-bit register. It is intended primarily for use in multiplication and division operations; however, it can also be used as an accumulator or holding register for some applications. The ALU output, F, can be loaded into the Q register, and/or the Q register can be selected as the source for the ALU S operand. The shifter at the input to the Q register provides the capability to shift the Q-register contents up one bit position (2Q) or down one bit position (Q/2). Only logical shifts are performed. QIO₀ and QIO₃ are bidirectional shift serial inputs/outputs. During a Q-register shift-up operation, QIO₀ is a serial shift input and QIO₃ is a serial shift output. During a shift-down operation, QIO₃ is a serial shift input and QIO₀ is a serial shift output.

Double-length arithmetic and logical shifting capability is provided by the Am2903. The double-length shift is performed by connection QIO₃ of the most significant slice to SIO₀ of the least significant slice, and executing an instruction which shifts both the ALU output and the Q register.

The Q register and shifter are controlled by the instruction inputs. Table 4 defines the Am2903 special functions and the operations which the Q register and shifter perform for each. When the Am2903 executes instructions other than the nine special functions, the Q register and shifter operation is controlled by instruction bits I₈I₇I₆I₅. Table 3 defines the Q register and shifter operation as a function of these four bits.

Output Buffers

The DB and Y ports are bidirectional I/O ports driven by three-state output buffers with external output enable controls. The Y output buffers are enabled when the \overline{OE}_Y input is LOW and are in the high-impedance state when \overline{OE}_Y is HIGH. Likewise, the DB output buffers are enabled when the \overline{OE}_B is LOW and in the high-impedance state when \overline{OE}_B is HIGH.

The zero, Z, pin is an open-collector input/output that can be wired ORed between slices. As an output it can be used as a zero detect status flag and generally indicates that the Y₀₋₃ pins are all LOW, whether they are driven from the Y output buffers or from an external source connected to the Y₀₋₃ pins. To some extent the meaning of this signal varies with the instruction being performed. Refer to Table 5 for an exact definition of this signal as a function of the Am2903 instruction.

Instruction Decoder

The Instruction Decoder generates required internal control signals as a function of the nine instruction inputs, I₀₋₈; the Instruction Enable input, \overline{IEN} ; the \overline{LSS} input; and the $\overline{WRITE}/\overline{MSS}$ input/output.

The \overline{WRITE} output is LOW when an instruction which writes data into the RAM is being executed. Refer to Tables 3 and 4 for a definition of the \overline{WRITE} output as a function of the Am2903 instruction inputs.

When \overline{IEN} is HIGH, the \overline{WRITE} output is forced HIGH and the Q register and Sign Compare Flip-Flop contents are preserved.

When \overline{IEN} is LOW, the \overline{WRITE} output is enabled and the Q register and Sign Compare Flip-Flop can be written according to the Am2903 instruction. The Sign Compare Flip-Flop is an on-chip flip-flop which is used during an Am2903 divide operation (see Fig. 3).

Programming the Am2903 Slice Position

Tying the \overline{LSS} input LOW programs the slice to operate as a least significant slice (LSS) and enables the \overline{WRITE} output signal onto

the $\overline{WRITE}/\overline{MSS}$ bidirectional I/O pin. When \overline{LSS} is tied HIGH, the $\overline{WRITE}/\overline{MSS}$ pin becomes an input pin. Tying the $\overline{WRITE}/\overline{MSS}$ pin HIGH programs the slice to operate as an intermediate slice (IS), and tying it LOW programs the slice to operate as a most significant slice (MSS).

Am2903 Special Functions

The Am2903 provides nine special functions which facilitate the implementation of the following operations:

- Single- and double-length normalization
- 2's complement division
- Conversion between 2's complement and sign magnitude representation
- Incrementation by 1 or 2

Table 4 defines these special functions.

The single-length and double-length normalization functions can be used to adjust a single-precision or double-precision floating-point number in order to bring its mantissa within a specified range.

Three special functions which can be used to perform a 2's complement, non-restoring divide operation are provided by the Am2903. These functions provide both single- and double-precision divide operations and can be performed in n clock cycles, where n is the number of bits in the quotient.

The unsigned multiply special function and the two 2's complement multiply special functions can be used to multiply two n -bit unsigned or 2's complement numbers in n clock cycles. These functions utilize the conditional add and shift algorithm. During the last cycle of the 2's complement multiplication, a conditional subtraction, rather than addition, is performed because the sign bit of the multiplier carries negative weight.

The sign/magnitude-2's complement special function can be used to convert number representation systems. A number expressed in sign/magnitude representation can be converted to the 2's complement representation, and vice-versa, in one clock cycle.

The increment by 1 and increment by 2 special functions can be used to increment an unsigned or 2's complement number by 1 or 2. This is useful in 16-bit-word, byte-addressable machines, where the word addresses are multiples of 2.

Pin Definitions

A ₀₋₃	Four RAM address inputs which contain the address of the RAM word appearing at the RAM A output port.
B ₀₋₃	Four RAM address inputs which contain the address of the RAM word appearing at the RAM B

Table 3 ALU Destination Control for I_0 OR I_1 OR I_2 OR I_3 OR I_4 = HIGH, \overline{IEN} = LOW

I_8	I_7	I_6	I_5	Hex code	ALU shifter function	SIO_3		Y_3	
						Most sig. slice	Other slices	Most sig. slice	Other slices
L	L	L	L	0	Arith. $F/2 \rightarrow Y$	Input	Input	F_3	SIO_3
L	L	L	H	1	Log. $F/2 \rightarrow Y$	Input	Input	SIO_3	SIO_3
L	L	H	L	2	Arith. $F/2 \rightarrow Y$	Input	Input	F_3	SIO_3
L	L	H	H	3	Log. $F/2 \rightarrow Y$	Input	Input	SIO_3	SIO_3
L	H	L	L	4	$F \rightarrow Y$	Input	Input	F_3	F_3
L	H	L	H	5	$F \rightarrow Y$	Input	Input	F_3	F_3
L	H	H	L	6	$F \rightarrow Y$	Input	Input	F_3	F_3
L	H	H	H	7	$F \rightarrow Y$	Input	Input	F_3	F_3
H	L	L	L	8	Arith. $2F \rightarrow Y$	F_2	F_3	F_3	F_2
H	L	L	H	9	Log. $2F \rightarrow Y$	F_3	F_3	F_2	F_2
H	L	H	L	A	Arith. $2F \rightarrow Y$	F_2	F_3	F_3	F_2
H	L	H	H	B	Log. $2F \rightarrow Y$	F_3	F_3	F_2	F_2
H	H	L	L	C	$F \rightarrow Y$	F_3	F_3	F_3	F_3
H	H	L	H	D	$F \rightarrow Y$	F_3	F_3	F_3	F_3
H	H	H	L	E	$SIO_0 \rightarrow Y_0, Y_1, Y_2, Y_3$	SIO_0	SIO_0	SIO_0	SIO_0
H	H	H	H	F	$F \rightarrow Y$	F_3	F_3	F_3	F_3

Parity = $F_3 \nabla F_2 \nabla F_1 \nabla F_0 \nabla SIO_3$

∇ = Exclusive OR

- \overline{WE} output port and into which new data is written when the \overline{WE} input and the CP input are LOW. The RAM write enable input. If \overline{WE} is LOW, data at the Y I/O port is written into the RAM when the CP input is LOW. When \overline{WE} is HIGH, writing data into the RAM is inhibited.
- DA_{0-3} A 4-bit external data input which can be selected as one of the Am2903 ALU operand sources; DA_0 is the least significant bit.
- \overline{EA} A control input which, when HIGH, selects DA_{0-3} and, when LOW, selects RAM output A as the ALU R operand.
- DB_{0-3} A 4-bit external data input/output. Under control of the $\overline{OE_B}$ input, RAM output port B can be directly read on these lines, or input data on these lines can be selected as the ALU S operand.
- $\overline{OE_B}$ A control input which, when LOW, enables RAM output B onto the DB_{0-3} lines and, when HIGH, disables the RAM output B tri-state buffers.

- C_n The carry-in input to the Am2903 ALU.
- I_{0-8} The nine instruction inputs used to select the Am2903 operation to be performed.
- \overline{IEN} The Instruction enable input which, when LOW, enables the WRITE output and allows the Q register and the Sign Compare Flip-Flop to be written. When \overline{IEN} is HIGH, the WRITE output is forced HIGH and the Q register and Sign Compare Flip-Flop are in the hold mode.
- C_{n+4} This output generally indicates the carry-out of the Am2903 ALU. Refer to Table 5 for an exact definition of this pin.
- \overline{G}/N A multi-purpose pin which indicates the carry generate, \overline{G} , function at the least significant and intermediate slices, and generally indicates the sign, N, of the ALU result at the most significant slice. Refer to Table 5 for an exact definition of this pin.
- \overline{P}/OVR A multi-purpose pin which indicates the carry

Y_2		Y_1	Y_0	SIO_0	\overline{Write}	Q Reg. & shifter function	QIO_3	QIO_0
Most sig. slice	Other slices							
SIO_3	F_3	F_2	F_1	F_0	L	Hold	Hi-Z	Hi-Z
F_3	F_3	F_2	F_1	F_0	L	Hold	Hi-Z	Hi-Z
SIO_3	F_3	F_2	F_1	F_0	L	Log. $Q/2 \rightarrow Q$	Input	Q_0
F_3	F_3	F_2	F_1	F_0	L	Log. $Q/2 \rightarrow Q$	Input	Q_0
F_2	F_2	F_1	F_0	Parity	L	Hold	Hi-Z	Hi-Z
F_2	F_2	F_1	F_0	Parity	H	Log. $Q/2 \rightarrow Q$	Input	Q_0
F_2	F_2	F_1	F_0	Parity	H	$F \rightarrow Q$	Hi-Z	Hi-Z
F_2	F_2	F_1	F_0	Parity	L	$F \rightarrow Q$	Hi-Z	Hi-Z
F_1	F_1	F_0	SIO_0	Input	L	Hold	Hi-Z	Hi-Z
F_1	F_1	F_0	SIO_0	Input	L	Hold	Hi-Z	Hi-Z
F_1	F_1	F_0	SIO_0	Input	L	Log. $2Q \rightarrow Q$	Q_3	Input
F_1	F_1	F_0	SIO_0	Input	L	Log. $2Q \rightarrow Q$	Q_3	Input
F_2	F_2	F_1	F_0	Hi-Z	H	Hold	Hi-Z	Hi-Z
F_2	F_2	F_1	F_0	Hi-Z	H	Log. $2Q \rightarrow Q$	Q_3	Input
SIO_0	SIO_0	SIO_0	SIO_0	Input	L	Hold	Hi-Z	Hi-Z
F_2	F_2	F_1	F_0	Hi-Z	L	Hold	Hi-Z	Hi-Z

L = LOW

Hi-Z = high-impedance

H = HIGH

propagate, \overline{P} , function at the least significant and intermediate slices, and indicates the conventional 2's complement overflow, OVR, signal at the most significant slice. Refer to Table 5 for an exact definition of this pin.

Z An open-collector input/output pin which, when HIGH, generally indicates the Y_{0-3} outputs are all LOW. For some special functions, Z is used as an input pin. Refer to Table 5 for an exact definition of this pin.

SIO_0 , SIO_3 Bidirectional serial shift inputs/outputs for the ALU shifter. During a shift-up operation, SIO_0 is an input and SIO_3 an output. During a shift-down operation, SIO_3 is an input and SIO_0 is an output. Refer to Tables 3 and 4 for an exact definition of these pins.

QIO_0 , QIO_3 Bidirectional serial shift inputs/outputs for the Q shifter which operate like SIO_0 and SIO_3 . Refer to Tables 3 and 4 for an exact definition of these pins.

 \overline{LSS} $\overline{WRITE}/$
MSS Y_{0-3}

An input pin which, when tied LOW, programs the chip to act as the least significant slice (LSS) of an Am2903 array and enables the \overline{WRITE} output onto the \overline{WRITE}/MSS pin. When \overline{LSS} is tied HIGH, the chip is programmed to operate as either an intermediate or most significant slice and the \overline{WRITE} output buffer is disabled.

When \overline{LSS} is tied LOW, the \overline{WRITE} output signal appears at this pin; the \overline{WRITE} signal is LOW when an instruction which writes data into the RAM is being executed. When \overline{LSS} is tied HIGH, \overline{WRITE}/MSS is an input pin; tying it HIGH programs the chip to operate as an intermediate slice (IS) and tying it LOW programs the chip to operate as the most significant slice (MSS).

Four data inputs/outputs of the Am2903. Under control of the \overline{OE}_Y input, the ALU shifter output data can be enabled onto these lines, or these lines can be used as data inputs when external data is written directly into the RAM.

Table 4

I_8	I_7	I_6	I_5	Hex code	Special function	ALU function	ALU shifter function	SIO_3		SIO_0	Q Reg. \rightarrow shifter function	QIO_3	QIO_0	<u>WRITE</u>
L	L	L	X	0, 1	Unsigned Multiply	$F = S + C_n$ if $Z = L$ $F = R + S + C_n$ if $Z = H$	Log. $F/2 \rightarrow Y$ (Note 1)	Hi-Z	Input	F_0	Log. $Q/2 \rightarrow Q$	Input	Q_0	L
L	L	H	X	2, 3	Two's Complement Multiply	$F = S + C_n$ if $Z = L$ $F = R + S + C_n$ if $Z = H$	Log. $F/2 \rightarrow Y$ (Note 2)	Hi-Z	Input	F_0	Log. $Q/2 \rightarrow Q$	Input	Q_0	L
L	H	L	L	4	Increment by One or Two	$F = S + 1 + C_n$	$F \rightarrow Y$	Input	Input	Parity	Hold	Hi-Z	Hi-Z	L
L	H	L	H	5	Sign/Magnitude-Two's Complement	$F = S + C_n$ if $Z = L$ $F = \bar{S} + C_n$ if $Z = H$	$F \rightarrow Y$ (Note 3)	Input	Input	Parity	Hold	Hi-Z	Hi-Z	L
L	H	H	X	6, 7	Two's Complement Multiply, Correction	$F = S + C_n$ if $Z = L$ $F = S - R - 1 + C_n$ if $Z = H$	Log. $F/2 \rightarrow Y$ (Note 2)	Hi-Z	Input	F_0	Log. $Q/2 \rightarrow Q$	Input	Q_0	L
H	L	L	X	8, 9	Single Length Normalize	$F = S + C_n$	$F \rightarrow Y$	F_3	F_3	Hi-Z	Log. $2Q \rightarrow Q$	Q_3	Input	L
H	L	H	X	A, B	Double Length Normalize and First Divide Op.	$F = S + C_n$	Log. $2F \rightarrow Y$	$R_3 \nabla F_3$	F_3	Input	Log. $2Q \rightarrow Q$	Q_3	Input	L
H	H	L	X	C, D	Two's Complement Divide	$F = S + R + C_n$ if $Z = L$ $F = S - R - 1 + C_n$ if $Z = H$	Log. $2F \rightarrow Y$	$\overline{R_3 \nabla F_3}$	F_3	Input	Log. $2Q \rightarrow Q$	Q_3	Input	L
H	H	H	X	E, F	Two's Complement Divide, Correction and Remainder	$F = S + R + C_n$ if $Z = L$ $F = S - R - 1 + C_n$ if $Z = H$	$F \rightarrow Y$	F_3	F_3	Hi-Z	Log. $2Q \rightarrow Q$	Q_3	Input	L

NOTES 1. At the most significant slice only, the C_{n+1} signal is internally gated to the Y_2 output.
 2. At the most significant slice only, $F_3 \nabla OVR$ is internally gated to the Y_3 output.
 3. At the most significant slice only, $S_3 \nabla F_3$ is generated at the Y_3 output.

Hi-Z = high impedance
 ∇ = Exclusive OR
 Parity = $SIO_3 \nabla F_3 \nabla F_2 \nabla F_1 \nabla F_0$

L = LOW
 H = HIGH
 X = don't care

Table 5

(Hx) $I_{A1}I_{A2}I_{A3}$	(Hx) $I_{A3}I_{A1}I_{A2}$	GI ($I = 0$ to 3)	PI ($I = 0$ to 3)	C_{n+4}	$\bar{P}OVR$		\bar{G}/N	Other slices	Most sig. slice	Z	Least sig. slice
					Most sig. slice	Other slices					
X	0	H	1	0	0	0	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	1	X	$\bar{R}_1 \vee S_1$	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	2	X	$R_1 \wedge \bar{S}_1$	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	3	X	$R_1 \wedge S_1$	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	4	X	0	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	5	X	0	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	6	X	0	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	7	X	0	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	8	X	0	0	0	0	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	9	X	$\bar{R}_1 \wedge S_1$	0	0	0	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	A	X	$R_1 \wedge S_1$	0	0	0	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	B	X	$\bar{R}_1 \wedge S_1$	0	0	0	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	C	X	$R_1 \wedge S_1$	0	0	0	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	D	X	$\bar{R}_1 \wedge \bar{S}_1$	0	0	0	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	E	X	$R_1 \wedge S_1$	0	0	0	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
X	F	X	$\bar{R}_1 \wedge \bar{S}_1$	0	0	0	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
0, 1	0	L	0 if Z = L $R_1 \wedge S_1$ if Z = H	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	Input	Input	Q_0
2, 3	0	L	0 if Z = L $R_1 \wedge S_1$ if Z = H	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	Input	Input	Q_0
4	0	L	See Note 1	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$	$\bar{Y}_0\bar{Y}_1\bar{Y}_2\bar{Y}_3$
5	0	L	0 if Z = L \bar{S}_1 if Z = H	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	S_3	Input	Input
6, 7	0	L	0 if Z = L $\bar{R}_1 \wedge S_1$ if Z = H	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	Input	Input	Q_0
8, 9	0	L	0	See Note 3	$Q_2 \nabla Q_1$	\bar{P}	\bar{G}	\bar{G}	$\bar{Q}_0\bar{Q}_1\bar{Q}_2\bar{Q}_3$	$\bar{Q}_0\bar{Q}_1\bar{Q}_2\bar{Q}_3$	$\bar{Q}_0\bar{Q}_1\bar{Q}_2\bar{Q}_3$
A, B	0	L	0	See Note 4	$F_2 \nabla F_1$	\bar{P}	\bar{G}	\bar{G}	See Note 5	See Note 5	See Note 5
C, D	0	L	$R_1 \wedge S_1$ if Z = L $\bar{R}_1 \wedge S_1$ if Z = H	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	Sign Compare FF Output	Input	Input
E, F	0	L	$R_1 \wedge S_1$ if Z = L $\bar{R}_1 \wedge S_1$ if Z = H	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	\bar{G}	\bar{G}	Sign Compare FF Output	Input	Input

Notes 1. if \bar{LSS} is LOW, $G_0 = S_0$ and $G_{1,2,3} = 0$ if \bar{LSS} is HIGH, $G_{0,1,2,3} = 0$ 2. if \bar{LSS} is LOW, $P_0 = 1$ and $P_{1,2,3} = S_{1,2,3}$ if \bar{LSS} is HIGH, $P_1 = S_1$ 3. At the most significant slice, $C_{n+4} = Q_3 \nabla Q_2$ At other slices, $C_{n+4} = G \vee PC_n$ 4. At the most significant slice, $C_{n+4} = F_3 \nabla F_2$ At other slices, $C_{n+4} = G \vee PC_n$ 5. $Z = \bar{Q}_0\bar{Q}_1\bar{Q}_2\bar{Q}_3\bar{F}_1\bar{F}_2\bar{F}_3$ $L = \text{LOW} = 0$ $H = \text{HIGH} = 1$ $\wedge = \text{OR}$ $\wedge = \text{AND}$ $\nabla = \text{EXCLUSIVE OR}$ $P = P_0P_1P_2P_3$ $G = G_1 \vee G_2P_3 \vee G_3P_2 \vee G_4P_1P_2P_3$ $C_{n+3} = G_2 \vee G_1P_2 \vee G_3P_1P_2 \vee C_0P_1P_2P_3$

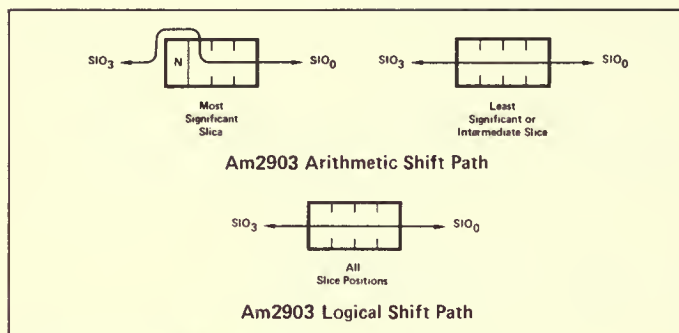


Fig. 3. Sign compare flip-flop.

- OE_Y** A control input which, when LOW, enables the ALU shifter output data onto the Y₀₋₃ lines and, when HIGH, disables the Y₀₋₃ three-state output buffers.
- SP** The clock input to the Am2903. The Q Register and Sign Compare Flip-Flop are clocked on the LOW-to-HIGH transition of the CP signal. When enabled by WE, data is written in the RAM when CP is LOW.

Using the Am2903

Am2903 Applications

The Am2903 is designed to be used in microprogrammed systems. Figure 4 illustrates a recommended architecture. The control and data inputs to the Am2903 normally will all come from registers clocked at the same time as the Am2903. The register inputs come from a ROM or PROM—the “microprogram store.” This memory contains sequences of microinstructions which apply the proper control signals to the Am2903’s and other circuits to execute the desired operation.

The address lines of the microprogram store are driven from the Am2910 Microprogram Sequencer. This device has facilities for storing an address, incrementing an address, jumping to any address, and linking subroutines. The Am2910 is controlled by some of the bits coming from the microprogram store. Essentially, these bits are the “next instruction” control.

Note that with the microprogram register in between the microprogram memory store and the Am2903’s, a microinstruction accessed on one cycle is executed on the next cycle. As one microinstruction is executed, the next microinstruction is being read from microprogram memory. In this configuration, system speed is improved because the execution time in the Am2903’s occurs in parallel with the access time of the microprogram store. Without the “pipeline register,” these two functions must occur serially.

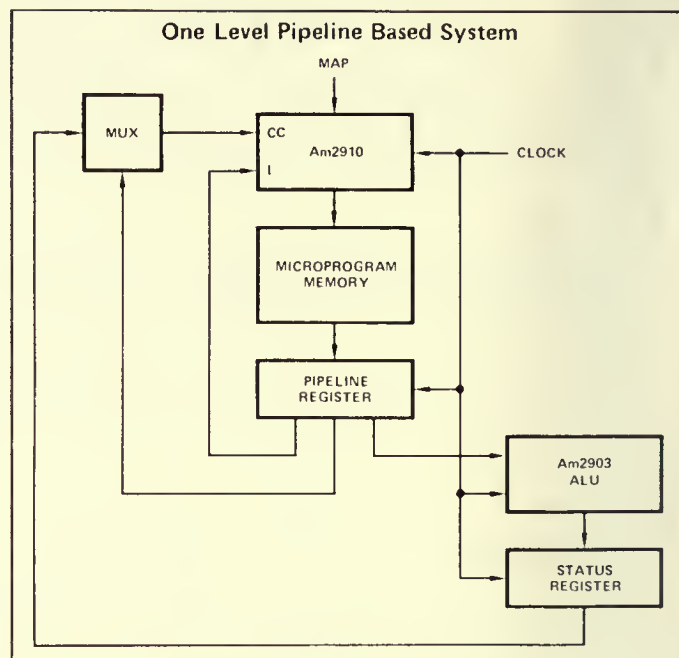


Fig. 4. Typical microprogram architecture.

Expansion of the Am2903

The Am2903 is a 4-bit CPU slice. Any number of Am2903’s can be interconnected to form CPU’s of 8, 16, 32, or more bits, in 4-bit increments. Figure 5 illustrates the interconnection of four Am2903’s to form a 16-bit CPU, using ripple carry.

With the exception of the carry interconnection, all expansion schemes are the same. The QIO₃ and SIO₃ pins are bidirectional left/right shift lines at the MSB of the device. For all devices except the most significant, these lines are connected to the QIO₀ and SIO₀ pins of the adjacent more significant device. These connections allow the Q registers of all Am2903’s to be shifted left or right as a contiguous *n*-bit register, and also allow the ALU output data to be shifted left or right as a contiguous *n*-bit word prior to storage in the RAM. At the LSB and MSB of the CPU, the shift pins should be connected to a shift multiplexer which can be controlled by the microcode to select the appropriate input signals to the shift inputs.

Device 1 has been defined as the least significant slice (LSS) and its LSS pin has accordingly been grounded. The Write/Most Significant Slice (WRITE/MSS) pin of device 1 is now defined as being the Write output, which may now be used to drive the write enable (WE) signal common to the four devices. Devices 2 and 3 are designated as intermediate slices and hence the LSS and WRITE/MSS pins are tied HIGH. Device 4 is designated the

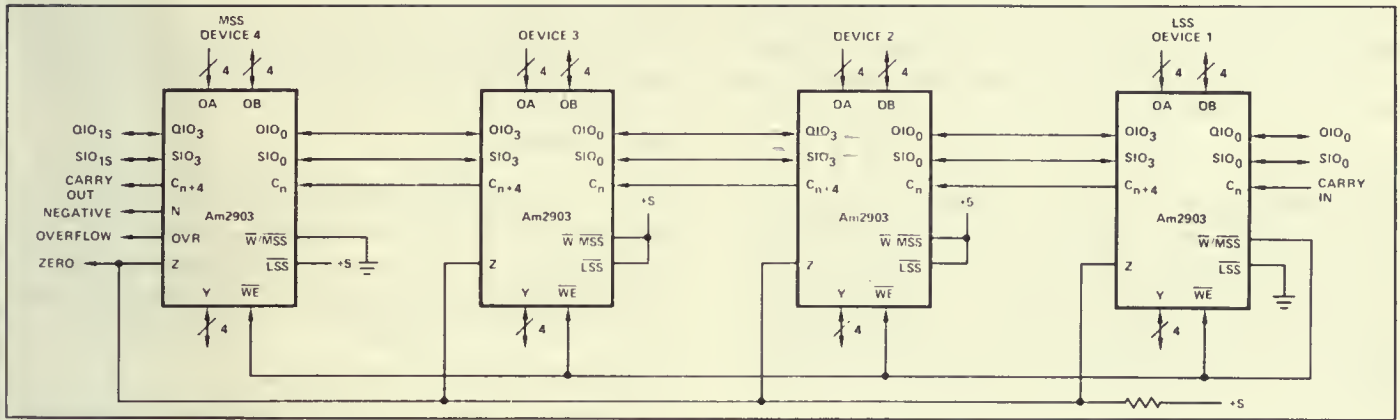


Fig. 5. 16-bit CPU with ripple carry.

most significant slice ($\overline{\text{MSS}}$) with the $\overline{\text{LSS}}$ pin tied HIGH and the $\overline{\text{WRITE/MSS}}$ pin held LOW. The open-collector, bidirectional Z pins are tied together for detecting zero or for inter-chip communication for some special instruction. The carry-out (C_{n+4}) is connected to the carry-in (C_n) of the next chip in the case of ripple carry. For a faster carry scheme, an AM2902 may be employed (as shown in Fig. 6) so that the $\overline{\text{G}}$ and $\overline{\text{P}}$ outputs of the Am2903 are connected to the appropriate $\overline{\text{G}}$ and $\overline{\text{P}}$ inputs of the Am2902, while the C_{n+x} , C_{n+y} , and C_{n+z} outputs of the Am2902 are connected to the C_n input of the appropriate Am2903. Note that $\overline{\text{G}}$, $\overline{\text{N}}$ and $\overline{\text{P}}$ /OVR pin functions are device-dependent. The most

significant slice outputs N and OVR while all other slices output $\overline{\text{G}}$ and $\overline{\text{P}}$.

The $\overline{\text{IEN}}$ pin of the Am2903 allows the option of conditional instruction execution. If $\overline{\text{IEN}}$ is LOW, all internal clocking is enabled, allowing the latches, RAM, and Q register to function, if $\overline{\text{IEN}}$ is HIGH, the RAM and Q register are disabled. The RAM is controlled by $\overline{\text{IEN}}$ if $\overline{\text{WE}}$ is connected to the $\overline{\text{WRITE}}$ output.

It would be appropriate at this point to mention that the Am2903 may be microcoded to work in either two- or three-address architecture modes. The two-address modes allow $A + B \rightarrow B$ while the three-address mode makes possible $A + B \rightarrow C$.

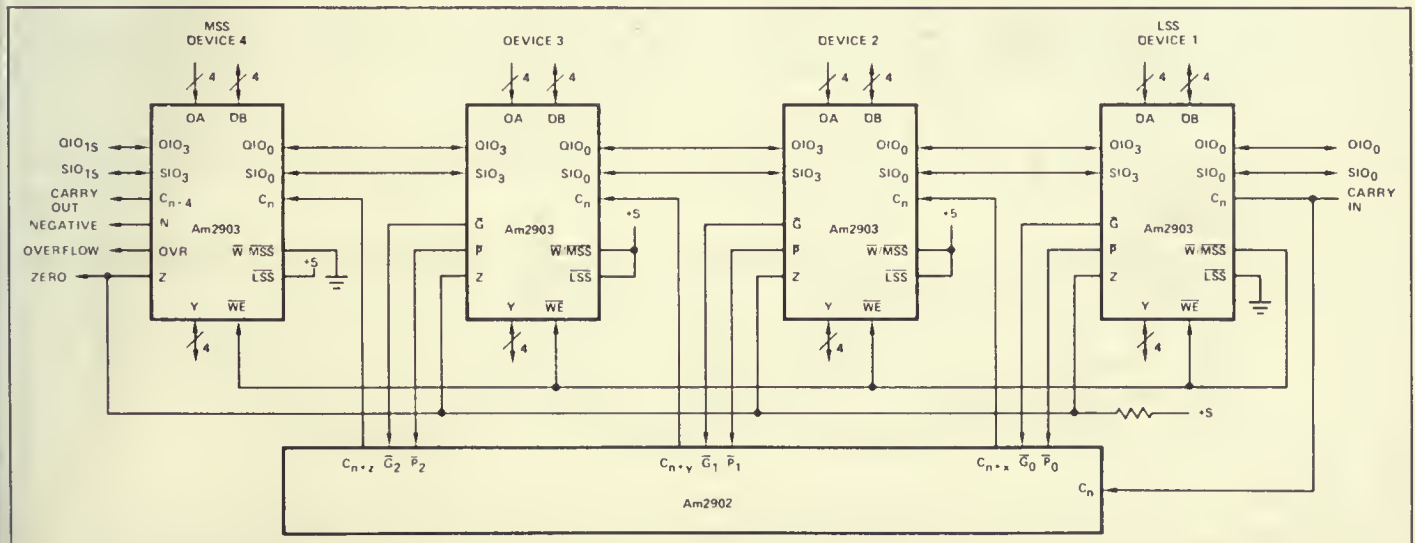


Fig. 6. 16-bit CPU with carry look ahead.

Implementation of a three-address architecture is made possible by varying the timing of \overline{IEN} in relationship to the external clock and changing the B address. This technique is discussed in more detail under Memory Expansion.

Parity

The Am2903 computes parity on a chosen word when the instruction bits I_{5-8} have the values of 4_{16} to 7_{16} as shown in Table 3. The computed parity is the result of the Exclusive-OR of the individual ALU outputs and SIO_3 . Parity output is found on SIO_0 . Parity between devices may be cascaded by the interconnection of the SIO_0 and SIO_3 ports of the devices as shown in Fig. 6. The equation for the parity output at the SIO_0 port of device 1 is given by $SIO_0 = F_{15} \vee F_{14} \vee F_{13} \vee \cdots \vee F_1 \vee F_0 \vee SIO_{15}$.

Sign Extend

Sign extend across any number of Am2903 devices can be done in one microcycle. Referring again to the table of instructions (Table 3), the sign extend instruction (Hex instruction E) on I_{5-8} causes the sign present at the SIO_0 port of a device to be extended across the device and appear at the SIO_3 port and at the Y outputs. If the least significant bit of the instruction (bit I_5) is HIGH, Hex instruction F is present on I_{5-8} , commanding a shifter pass instruction. At this time, F_3 of the ALU is present on the SIO_3 output pin. It is then possible to control the extension of the sign across chip boundaries by controlling the state of I_5 when I_{6-8} are HIGH. Figure 7 outlines the Am2903 in sign extend mode. With I_{6-8} held HIGH, the individual chip sign extend is controlled by I_{5A-D} . If, for example, I_{5A} and I_{5B} are HIGH while I_{5C} and I_{5D} are LOW, the signal present at the boundaries of devices 2 and 3 (F_3 of device 2) will be extended across devices 3 and 4 at the SIO_3 pin of device 4. The outputs of the four devices will be available at

their respective Y data ports. The next positive edge of the clock will load the Y outputs into the address selected by the B port. Hence, the results of the sign extension are stored in the RAM.

Special Functions

When $I_{0-4} = 0$, the Am2903 is in the special function mode. In this mode, both the source and destination are controlled by I_{5-8} . The special functions are in essence special microinstructions that are used to reduce the number of microcycles needed to execute certain functions in the Am2903.

Normalization, Single- and Double-Length

Normalization is used as a means of referencing a number to a fixed radix point. Normalization strips out all leading sign bits such that the two bits immediately adjacent to the radix point are of opposite polarity.

Normalization is commonly used in such operations as fixed-to-floating point conversion and division. The Am2903 provides for normalization by using the Single-Length and Double-Length Normalize commands. Figure 8a represents the Q register of a 16-bit processor which contains a positive number. When the Single-Length Normalize command is applied, each positive edge of the clock will cause the bits to shift toward the most significant bit (bit 15) of the Q register. Zeros are shifted in via the QIO_0 port. When the bits on either side of the radix point (bits 14 and 15) are of opposite value, the number is considered to be normalized, as shown in Fig. 8b. The event of normalization is externally indicated by a HIGH level on the C_{n+4} pin of the most significant slice ($C_{n+4} \text{ MSS} = Q_3 \text{ MSS} \vee Q_2 \text{ MSS}$).

There are also provisions made for a normalization indication via the OVR pin one microcycle before the same indication is

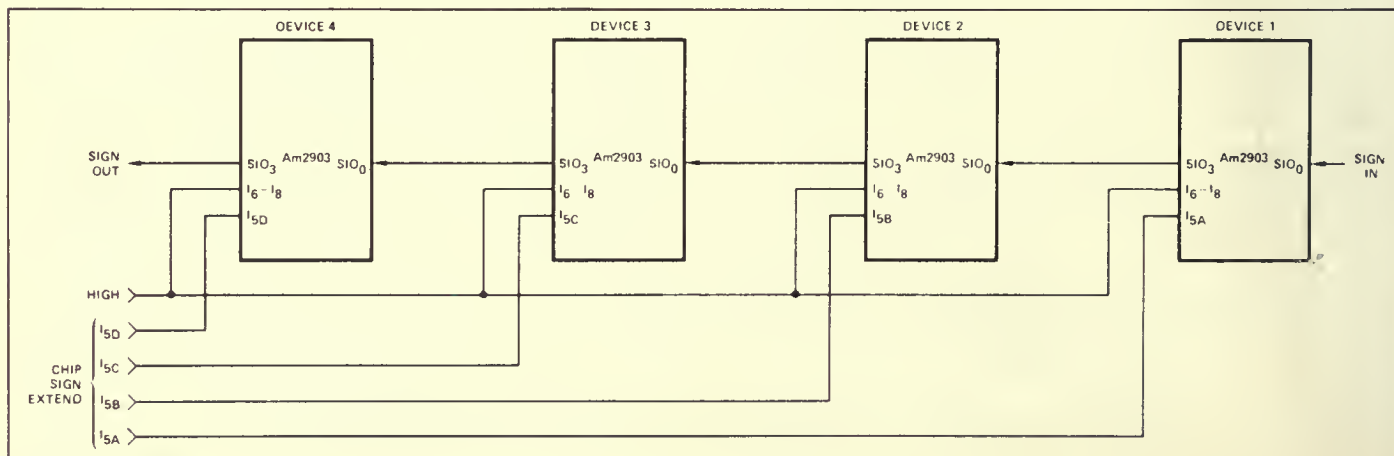


Fig. 7. Sign extend.

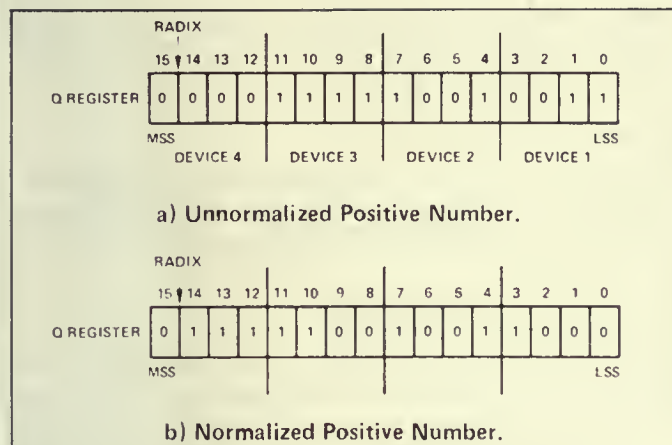


Fig. 8

available on the C_{n+4} pin ($OVR = Q_2 \text{ MSS} \vee Q_1 \text{ MSS}$). This is for use in applications that require a stage of register buffering of the normalization indication.

Since a number consisting of all zeros is not considered for normalization, the Am2903 indicates when such a condition arises. If the Q register is zero and the Single-Length Normalization command is given, a HIGH level will be present on the Z line. The sign output, N, indicates the sign of the number stored in the Q register, Q₃ MSS. An unnormalized negative number (Fig. 9a) is normalized in the same manner as a positive number. The results of single-length normalization are shown in Fig. 9b. The device interconnection for single-length normalization is

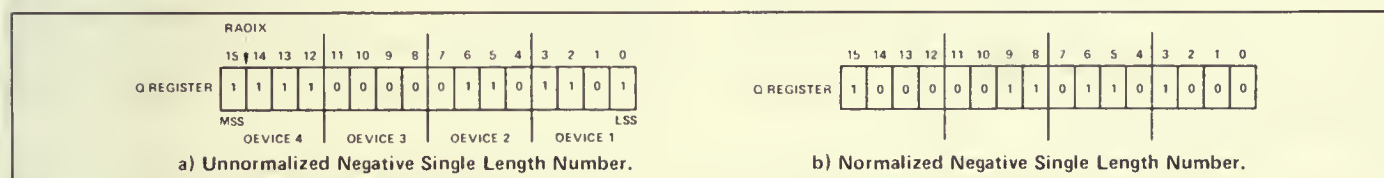


Fig. 9

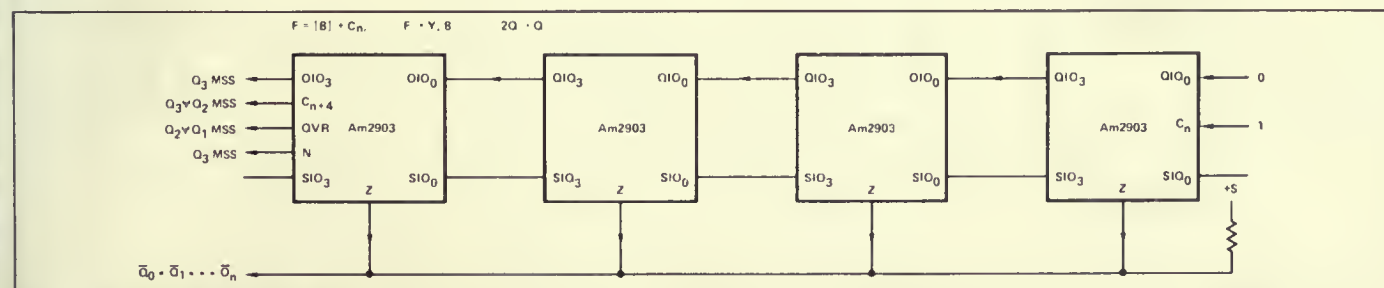


Fig. 10. Single-length normalize.

outlined in Fig. 10. During single-length normalization, the number of shifts performed to achieve normalization can be counted and stored in one of the working registers. This can be achieved by forcing a HIGH at the C_n input of the least significant slice, since during this special function the ALU performs the function $[B] + C_n$ and the result is stored in B.

Normalizing a double-length word can be done with the Double-Length Normalize command, which assumes that a user-selected RAM register contains the most significant portion of the word to be normalized while the Q register holds the least significant half (Fig. 11). The device interconnection for double-length normalization is shown in Fig. 12. The C_{n+4} , OVR, N, and Z outputs of the most significant slice perform the same functions in double-length normalization as they did in single-length normalization except that C_{n+4} , OVR, and N are derived from the output of the ALU of the most significant slice in the case of double-length normalization, instead of the Q register of the most significant slice as in single-length normalization. A high-level Z line in double-length normalization reveals that the outputs of the ALU and Q register are both zero, hence indicating that the double-length word is zero.

When double-length normalization is being performed, shift counting is done either with an extra microcycle or with an external counter.

Sign/Magnitude-2's Complement Conversion

As part of the special instruction set, the Am2903 can convert between 2's complement and sign/magnitude representations. Figure 13 illustrates the interconnection needed for sign/mag-

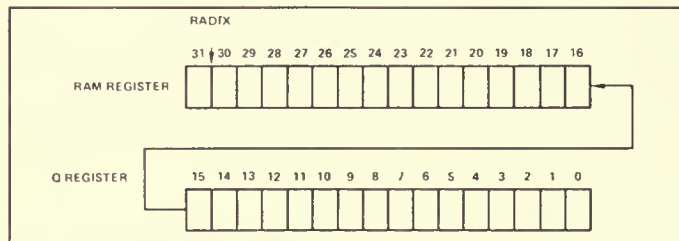


Fig. 11. Double-length word.

nitude-2's complement conversion. The C_n input of device 1 is connected to the Z pin. The sign bit (S_3 MSS) is brought out on the Z line and informs the other ALU's whether the conversion is being performed on a negative or a positive number. If the number attempted to be converted is the most negative number in 2's complement [i.e., $100 \dots 00(-2^n)$], an overflow indication will occur. This is because -2^n is 1 greater than any number that can be represented in sign magnitude notation and hence an attempted conversion to sign magnitude from -2^n will cause an

overflow. When minus zero in sign/magnitude notation ($100 \dots 0$) is converted to 2's complement notation, the correct result is obtained ($0 \dots 0$).

Increment by 1 or 2

Incrementation by 1 or 2 is made possible by the special function of the same name. This command is quite useful in the case of byte-addressable words. A word may be incremented by 1 if C_n is LOW or incremented by 2 if C_n is HIGH.

Unsigned Multiply

This special function allows for easy implementation of unsigned multiplication. Figure 14 is the multiply flow chart. The algorithm dictates that initially the RAM word addressed by address port B be zero, the multiplier be in the Q register, and the multiplicand be in the register addressed by address port A. The initial conditions for the execution of the algorithm are that (1) register R_0 be reset to zero; (2) the multiplicand be in R_1 ; and (3) the multiplier be in R_2 . The first operation transfers the multiplier R_2 to the Q register. The Unsigned Multiply (2's complement

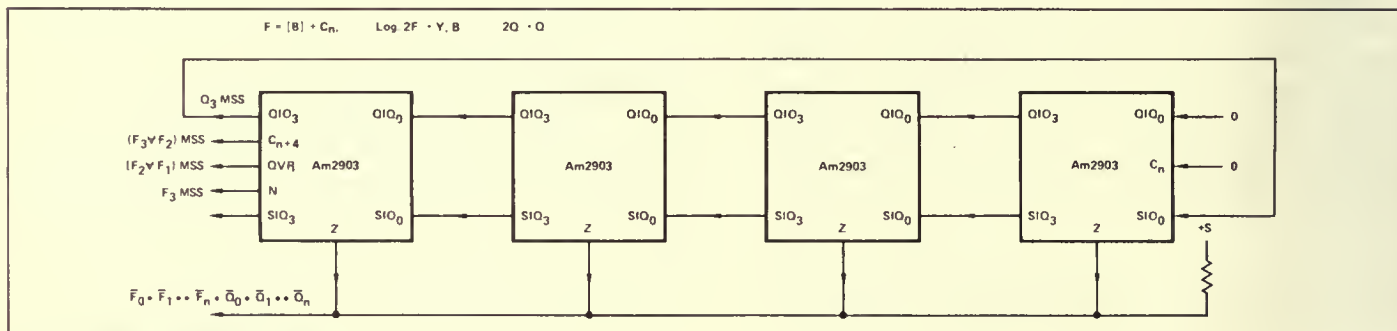
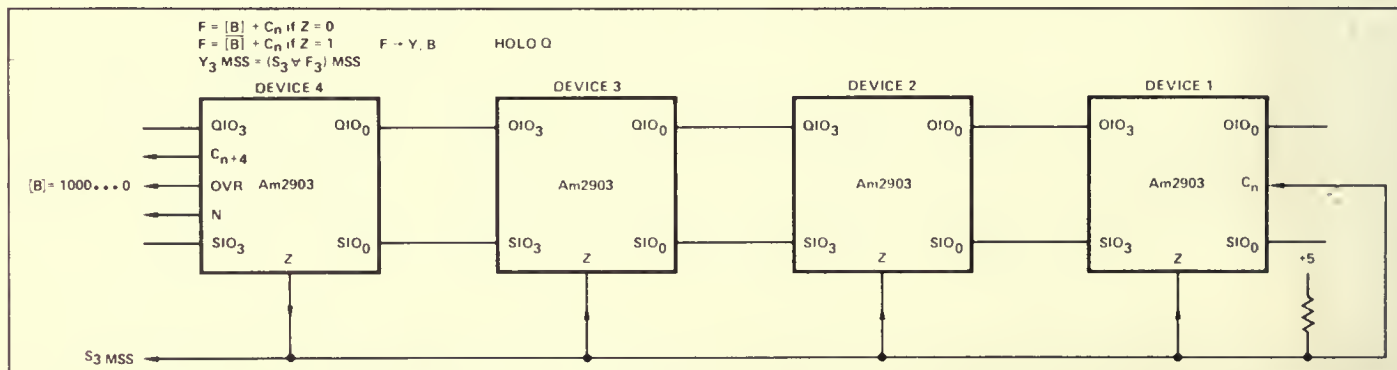


Fig. 12. Double-length normalize.

Fig. 13. 2's complement \leftrightarrow sign/magnitude.

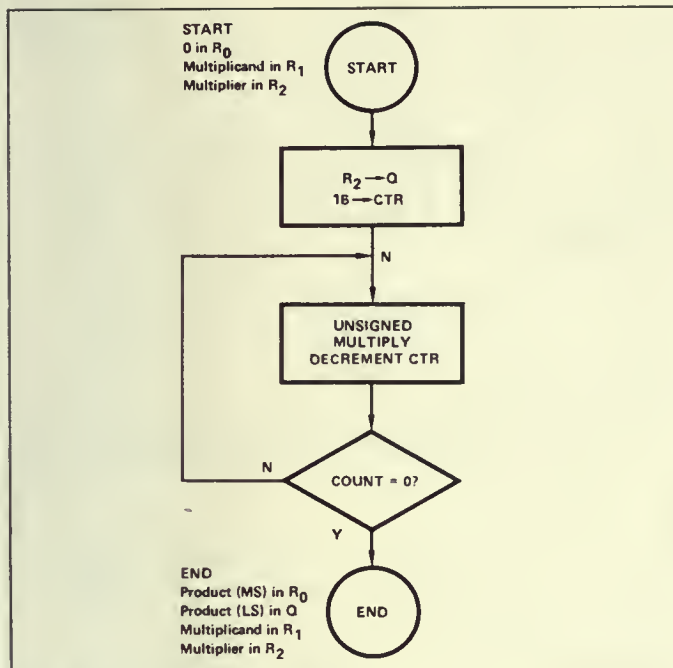


Fig. 14. 16x16 multiply flowchart.

multiply) instruction is then executed 16 (15) times. During the Multiply instruction, R_0 is addressed by RAM address port B and the multiplicand is addressed by RAM address port A.

When the Unsigned Multiply command is given, the Z pin of device 1 becomes an output while the Z pins of the remaining devices are specified as inputs as shown in Fig. 15. The Z output of device 1 is the same state as the least significant bit of the Q register during the Unsigned Multiply instruction; therefore, the

Z output of device 1 informs the ALU's of all the slices, via their Z pins, to output the sum of the partial product (referenced by the B address port) plus the multiplicand (referenced by the A address port) if $Z = 1$. If $Z = 0$, the output of the ALU is simply the partial product (referenced by the B address port). Since C_n is held LOW, it is not a factor in the computation. Each positive-going edge of the clock will internally shift the ALU outputs toward the least significant bit and simultaneously store the shifted results in the register selected by the B address port, thus becoming the new partial sum. During the down-shifting process, the C_{n+4} generated in device 4 is internally shifted into the Y_3 position of device 4. At this time, one bit of the multiplier will down-shift out of the QIO_0 ports of each device into the QIO_3 port of the next least significant slice. The partial product is shifted down between chips in a like manner, between the SIO_0 and SIO_3 ports, with SIO_0 of device 1 being connected to QIO_3 of device 4 for purposes of constructing a 32-bit-long register to hold the 32-bit product. At the finish of the 16×16 multiply, the most significant 16 bits of the product will be found in the registers referenced by the B address lines while the least significant 16 bits are stored in the Q register. Using a typical computer control unit (CCU), as shown in Fig. 16, the unsigned multiply operation requires only two lines of microcode, as shown in Fig. 17, and is executed in 17 microcycles.

2's Complement Multiplication

The algorithm for 2's complement multiplication is illustrated by Fig. 14. The initial conditions for 2's complement multiplication are the same as for the unsigned multiply operation. The 2's Complement Multiply command is applied for 15 clock cycles in the case of 16×16 multiply. During the down-shifting process the term N-OVR generated in device 4 is internally shifted into the Y_3 position of device 4. The data flow shown in Fig. 16 is

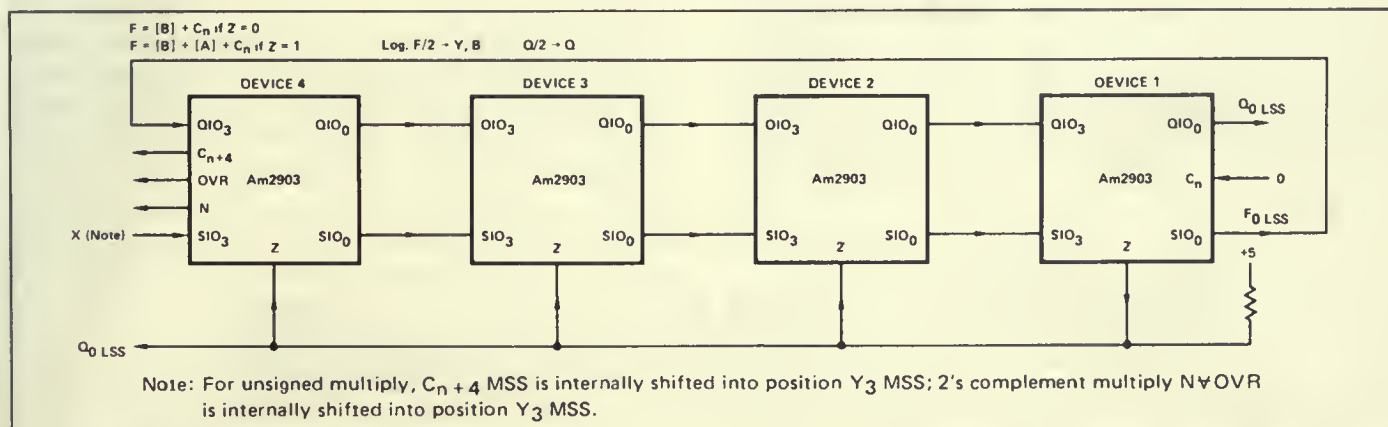


Fig. 15. Multiply.

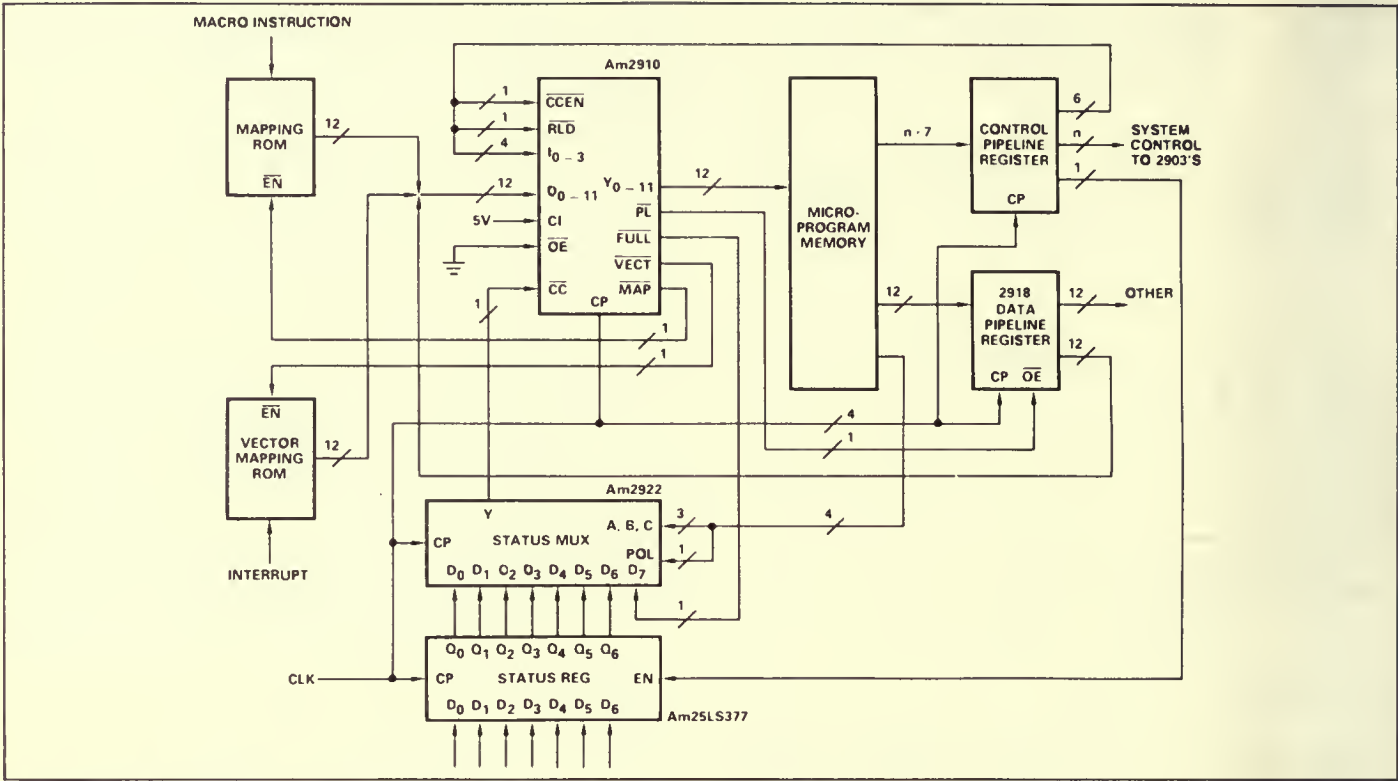


Fig. 16. Typical computer control unit (CCU).

still valid. After 15 cycles, the sign bit of the multiplier is present at the Z output of device 1. At this time, the user must place the 2's Complement Multiply Last Cycle command on the instruction lines. The interconnection for this instruction is shown in Fig. 18. On the next positive edge of the clock, the Am2903 will adjust the partial product, if the sign of the multiplier is negative, by subtracting out the 2's complement representation of the multiplicand. If the sign bit is positive, the partial product is not adjusted. At this point, 2's complement multiplication is complete. Using a typical CCU, the 2's complement multiply operation requires

only three lines of microcode, as shown in Fig. 19, and is executed in 17 microcycles.

2's Complement Division

The division process is accomplished by using a four-quadrant non-restoring algorithm which yields an algebraically correct answer such that the divisor times the quotient plus the remainder equals the dividend. The algorithm works for both single-precision and multi-precision divide operations. The only condi-

Micro Memory Address	Am2910 Inst	Data Pipeline Reg.	I ₀	I _{4-I₁}	I _{8-I₅}	$\overline{\text{OEB}}$	$\overline{\text{OEY}}$	A _{3-A₀}	B _{3-B₀}	C _n	Comment
n	LDCT	00F ₁₆	X	6	6	X	X	R ₂	X	0	Load Counter & R ₂ → Q
n+1	RPCT	n+1	0	0	0	0	0	R ₁	R ₀	0	Unsigned Multiply

Fig. 17. Microcode for unsigned 16x16 multiply.

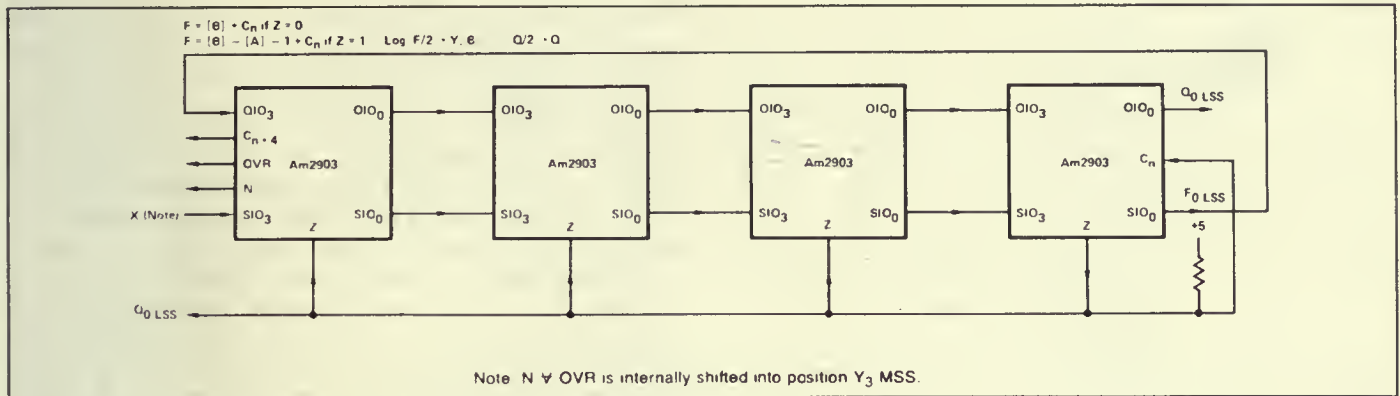


Fig. 18. 2's complement multiply, correction.

Memory Address	Am2910 Inst	Data Pipeline Reg.	Q ₀	Q ₁	Q ₂	Q ₃	Q ₄	Q ₅	Q ₆	Q ₇	Q ₈	Q ₉	Q ₁₀	Q ₁₁	Q ₁₂	Q ₁₃	Q ₁₄	Q ₁₅	Comment
n	LDCT	00E ₁₆	X	6	6	X	X	R ₂	X	0	0	0	0	0	0	0	0	0	Load Counter & R ₂ → Q
n+1	RPCT	n+1	0	0	2	0	0	R ₁	R ₀	0	0	0	0	0	0	0	0	0	2's Complement Multiply
n+2	X	X	0	0	6	0	0	R ₁	R ₀	Z	0	0	0	0	0	0	0	0	2's Complement Multiply (Last Cycle)

Fig. 19. Microcode for 2's complement 16x16 multiply.

tion that needs to be met is that the absolute magnitude of the divisor be greater than the absolute magnitude of the dividend. For multi-precision divide operations the least significant bit of the dividend is truncated. This is necessary if the answer is to be algebraically correct. Bias correction is automatically provided by forcing the least significant bit of the quotient to a 1, yet an algebraically correct answer is still maintained. Once the algorithm is completed, the answer may be modified to meet the user's formal requirements, such as rounding off or converting the remainder so that its sign is the same as the dividend's. These format modifications are accomplished using the standard Am2903 instructions.

The true value of the remainder is equal to the value stored in the working register 2^{n-1} when n is the number of quotient digits.

The following paragraphs describe a double-precision divide operation. The double-precision flow chart is based upon the use of the architecture detailed in Fig. 18.

Referring to the flow chart outlined in Fig. 20, we begin the algorithm with the assumption that the divisor is contained in R_0 , while the most significant and least significant halves of the dividend reside in R_1 and R_4 , respectively. The first step is to duplicate the divisor by copying the contents of R_0 into R_3 . Next

the most significant half of the dividend is copied by transferring the contents of R_1 into R_2 while simultaneously checking to ascertain if the divisor (R_0) is zero. If the divisor is zero then division is aborted. If the divisor is not zero, the copy of the most significant half of the dividend in R_2 is converted from its 2's complement to its sign/magnitude representation. The divisor in R_3 is converted in like manner in the next step, while a test is done to see if the results of the dividend conversion yielded an indication on the overflow pin of the Am2903. If the output of the overflow pin is a 1 then the dividend is -2^n and hence is the largest possible number, meaning that it cannot be less than the divisor. What must be done in this case is to scale the dividend by down-shifting the upper and lower halves stored in R_1 and R_4 respectively. After scaling, the routine requires that the algorithm be reinitiated at the beginning.

Conversely, if the output of the overflow pin is not a 1, the sign/magnitude representation of the divisor (R_3) is shifted up in the Am2903, removing the sign while at the same time testing the results of 2's complement to sign/magnitude conversion of the divisor in the Am2910. If the results of the test indicate that the divisor is -2^n , i.e., overflow equals 1, then the lower half of the dividend is placed in the Q register and division may proceed.

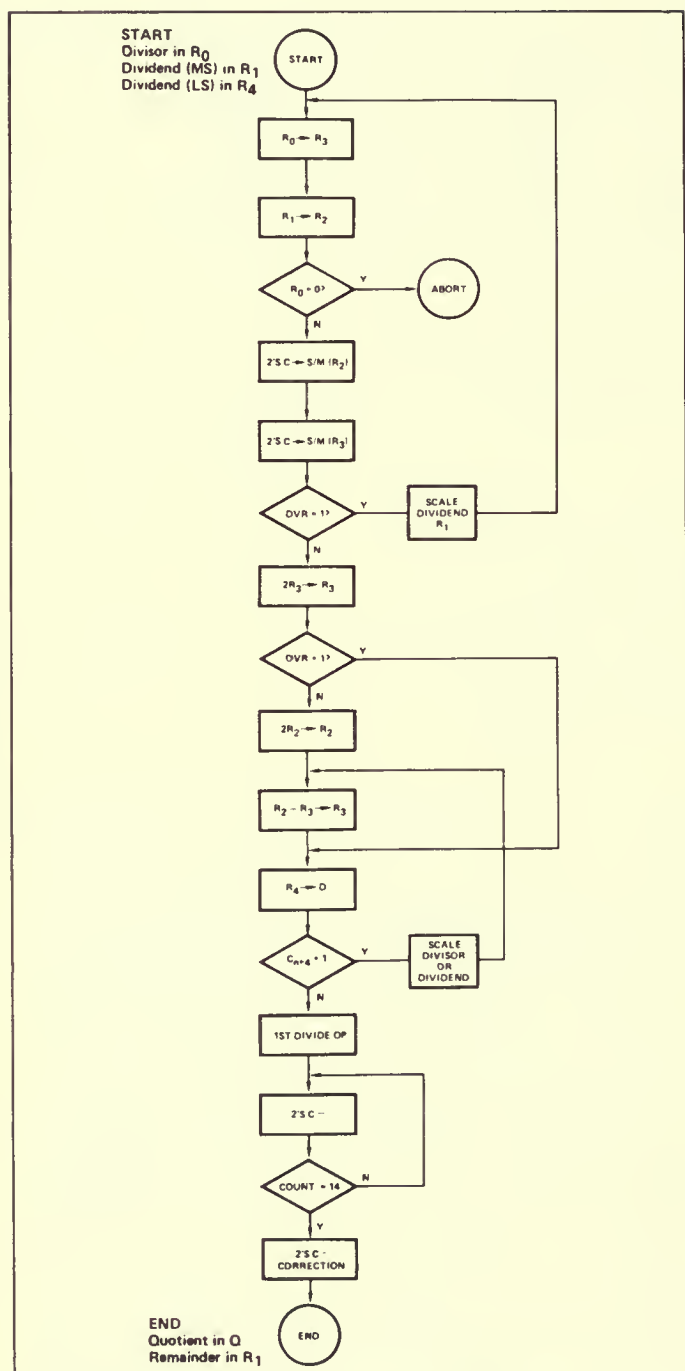


Fig. 20. Division flow chart—double precision divide.

This is possible because the divisor is now guaranteed to be greater than the dividend. If overflow is not a 1 then we must proceed by shifting out the sign of the sign/magnitude representation of the dividend stored in R_2 . At this point we are able to check whether the divisor is greater than the dividend by subtracting the absolute value of the divisor (R_3) from the absolute value of the upper half of the dividend (R_2) and storing the results in R_3 . Next, the least significant half of the dividend is transferred from R_4 to the Q register while simultaneously the carry from the result of the divisor-dividend subtraction is tested. If the carry (C_{n+4}) is 1, indicating the divisor is not greater than the dividend, then a scaling operation must occur. This involves either shifting up the divisor or shifting down the dividend. If the carry is not 1 then the divisor is greater than the dividend and division may now begin.

The first divide operation is used to ascertain the sign bit of the quotient. The 2's Complement Divide instruction is then executed 14 times in the case of a 16-bit divisor and a 32-bit dividend. The final step is the 2's Complement Correction command, which adjusts the quotient by allowing the least significant bit of the quotient to be set to a 1. At the end of the division algorithm the 16-bit quotient is found in the Q register while the remainder now replaces the most significant half of the dividend in R_1 . It should be noted that the remainder must be shifted down 15 places to represent its true value. The interconnections for these instructions are shown in Figs. 21, 22, 23. Using a typical CCU as shown in Fig. 15, the double-precision divide operation requires only 11 lines of microcode, as shown in Fig. 24.

For those applications that require truncation instead of bias correction, the same algorithm as above should be implemented except one additional 2's Complement Divide instruction should be used in lieu of the 2's Complement Divide Correction and Remainder instruction. However, this technique results in an invalid remainder.

It is possible to do multiple-precision divide operations beyond the double-precision divide shown above. For example, to do a triple-precision divide for a 16-bit CPU, the upper two-thirds of the dividend are stored in R_1 and Q as in the case for double-precision divide. The lower third of the dividend is stored in a scratch register, R_5 . After checking that the magnitude of the divisor is greater than the magnitude of the dividend, using the same tests as defined in Fig. 20, the procedure is as follows:

- 1 Execute a Double-Length Normalize/First Divide Operation instruction.
- 2 Execute the 2's Complement Divide instruction 15 times.
- 3 Transfer the contents of Q, the most significant half of the quotient, to R_2 .
- 4 Transfer R_5 to Q.
- 5 Execute the 2's Complement Divide instruction 15 times.

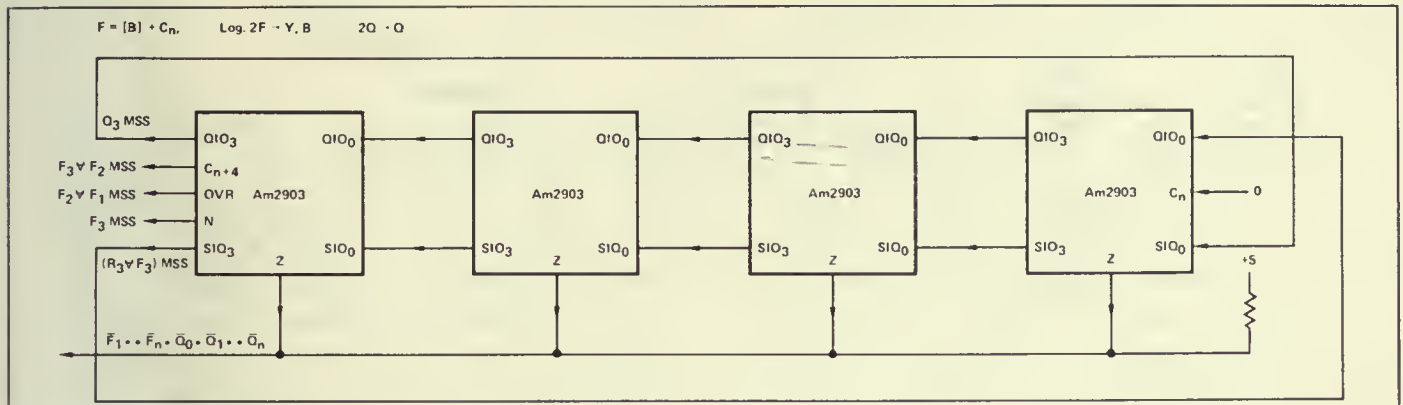


Fig. 21. Double-length normalize/first divide operation.

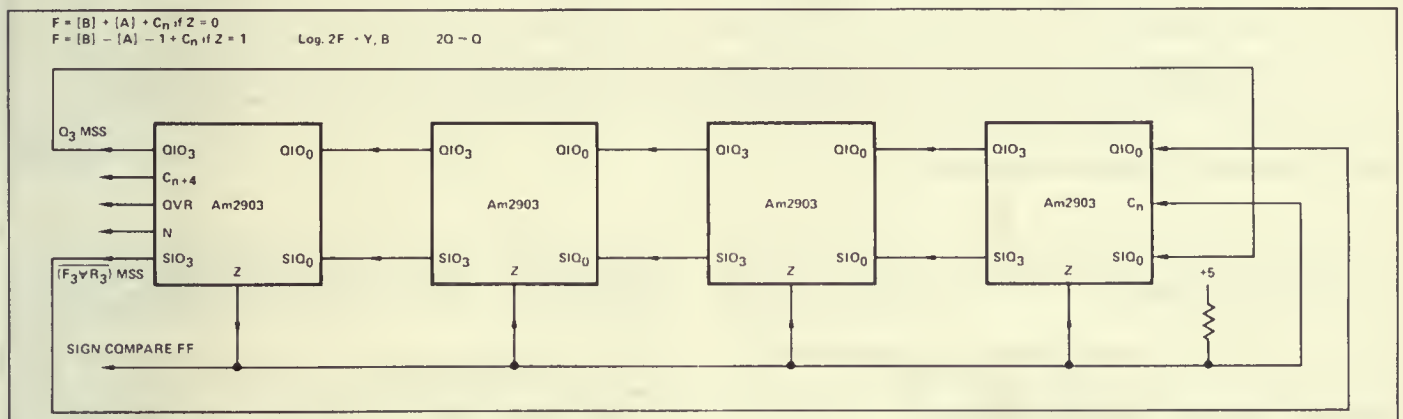


Fig. 22. 2's complement divide.

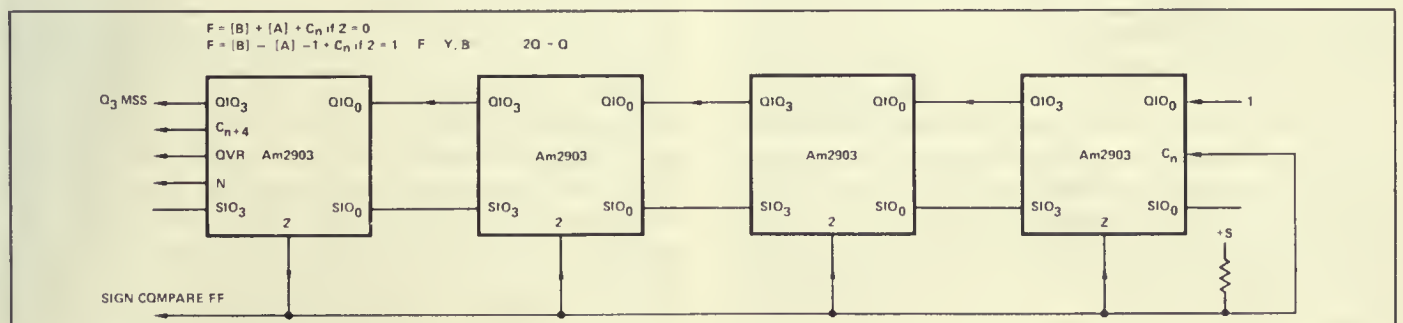


Fig. 23. 2's complement divide correction.

Micro Memory Address	Am2910 Inst.	Data Pipeline Reg.	Am2903							Am2922		Am29LS18 E		Comment
			I ₀	I ₄ -I ₁	I ₈ -I ₅	EA	A ₃ -A ₀	B ₃ -B ₀	C _n	SEL	POL			
n	CONT	X	0	6	4	0	R ₀	R ₃	0	X ₀	X	0	R ₀ → R ₃	
n+1	CJP	Abort	0	6	4	0	R ₁	R ₂	0	Z	1	X	R ₁ → R ₂ , if R ₀ = 0 Abort	
n+2	CONT	X	0	0	5	X	X	R ₂	0	X	X	0	2's C to S/M (R ₂)	
n+3	CJP	Scale Dividend	0	0	5	X	X	R ₃	0	OVR	1	0	2's C to S/M (R ₃), if OVR ≥ 1, scale	
n+4	CJP	n+7	0	4	9	X	X	R ₂	0	OVR	1	X	Shift out sign of divisor	
n+5	CONT	X	0	4	9	X	X	R ₃	0	X	X	X	Shift out sign of divisor	
n+6	CONT	X	0	2	F	0	R ₂	R ₃	1	X	X	0	Dividend - Divisor → R ₃	
n+7	CJP	Scale Dividend or Divisor	0	6	6	0	R ₄	X	0	C _{n+4}	0	X	R ₄ → Q, if Carry = 1, scale	
n+8	PUSH	00D ₁₆	0	0	A	0	R ₀	R ₁	0	0	1	X	Loop set up & First Divide Operation	
n+9	RFCT	X	0	0	C	0	R ₀	R ₁	Z	X	X	X	Test Loop Count & 2's C Divide	
n+A	CONT	X	0	0	E	0	R ₀	R ₁	Z	X	X	X	2's C Divide Correction	

Fig. 24. Microcode for double precision divide.

- 6 Execute the 2's Complement Divide Correction and Remainder instruction.

The upper half of the quotient is then in R₂, the lower half of the quotient is in Q, and the remainder is in R₁. This technique can be expanded for any precision which is required.

Byte Swap

The multi-port architecture of the Am2903 allows for easy implementation of high- and low-order byte swapping. Figure 25 outlines a byte-swap implementation utilizing two data ports. Initially, the lower-order 8-bit byte is stored in devices 1 and 2 while the high-order byte is in devices 3 and 4. When the user wishes to exchange the two bytes, the register location of the desired word is placed on the B address port. When the byte-swap line is brought LOW, the bytes to be swapped will be flowing from the DB ports of the Am2903 through the Am25LS240/244 three-state buffers. The outputs of the three-state buffers are permuted so that the byte swap is achieved. The resultant permuted data is presented to the DA ports of the Am2903, where it is reloaded into the memories of the Am2903 on the next positive edge of CP using the permuted data source and function

commands of $F = \bar{A}$ plus C_n(C_n = 0) for the Am25LS240 or $F = A$ plus C_n(C_n = 0) for the Am25LS244 and the destination command $F \rightarrow Y, B$.

A higher-speed technique for achieving the byte-swap operation uses the Y input/output ports with OE_Y held HIGH rather than the DA port inputs. This technique bypasses the ALU, thus allowing faster operation. The Am2903 destination command $F \rightarrow Y, B$ should be used.

Memory Expansion

The Am2903 allows for a theoretically infinite memory expansion. Figure 26 pictures a 4-bit slice of a system which has 48 words of RAM and 16 words of ROM. RAM storage is provided by the Am2903 and the Am29705's. The 29705 RAM is functionally identical to the Am2903 RAM. The Am29751 is used to store constants and masks and is addressable from address port A only. The system is organized around five data buses. Inter-bus communication may be done through the Am29705's or the Am2903. The memory addressing scheme specifies the data source for the R input of the ALU emanating from the register locations specified by address field A. A₀₋₃ address 16 memory locations in each chip while address bits A₄₋₆ are decoded and used

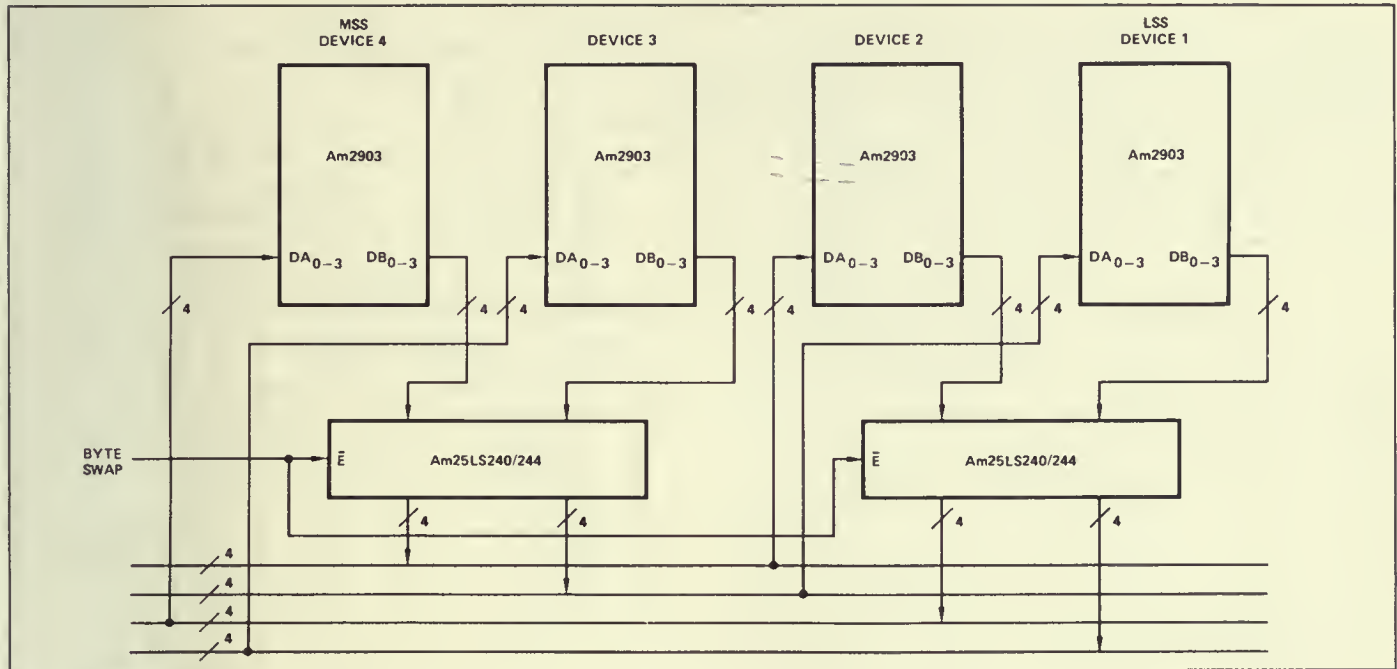


Fig. 25. Byte swap.

for the output enable for the desired chip. The B address field is used both to select the S input of the ALU and to specify the register location where the result of the ALU operation is to be stored.

Bits B₀₋₃ are for source register addressing in each chip. Bits B₄ and B₅ are used for chip output enable selection. B₆₋₉ access the 16 destination addresses on each chip, while bits B₁₀ and B₁₁ control the Write Enable of the desired chip. The source and destination register address are multiplexed so that when the clock is HIGH, the source register address is presented to the B address ports of the RAM's. The Instruction Enable (\bar{IEN}) is HIGH at this time. The data flows from the Y port or the internal B port, as selected by the decoder whose inputs are B₄ and B₅. When the clock goes LOW, the data emanating from the selected Y outputs of the Am29705's and the RAM outputs of the Am2903 are latched and the destination address is now selected for use by the RAM address lines. When the destination address stabilizes on the address lines, the \bar{IEN} pin is brought LOW. The \overline{WRITE} output of the Am2903 will now go LOW, enabling the decoder sourced by address bits B₁₀ and B₁₁. The selected decoder line will go LOW, allowing the desired memory location to be written into. To switch between two- and three-address architecture, the user

simply makes the source and destination addresses the same, i.e., B₀₋₃ = B₆₋₉. For two-address architecture, the MUX is removed from the circuit.

General Description of the Am2910

The Am2910 microprogram controller is an address sequencer intended for controlling the sequence of execution of microinstructions stored in microprogram memory. Besides the capability of sequential access, it provides conditional branching to any microinstruction within its 4096-microword range. A last-in, first-out stack provides microsubroutine return linkage and looping capability; there are five levels of nesting of microsubroutines. Microinstruction loop-count control is provided with a count capacity of 4096.

During each microinstruction, the microprogram controller provides a 12-bit address from one of four sources: (1) the microprogram address register (μPC), which usually contains an address 1 greater than the previous address; (2) an external (direct) input (D); (3) a register/counter (R) retaining data loaded during a previous microinstruction; or (4) a five-deep last-in, first-out stack (F).

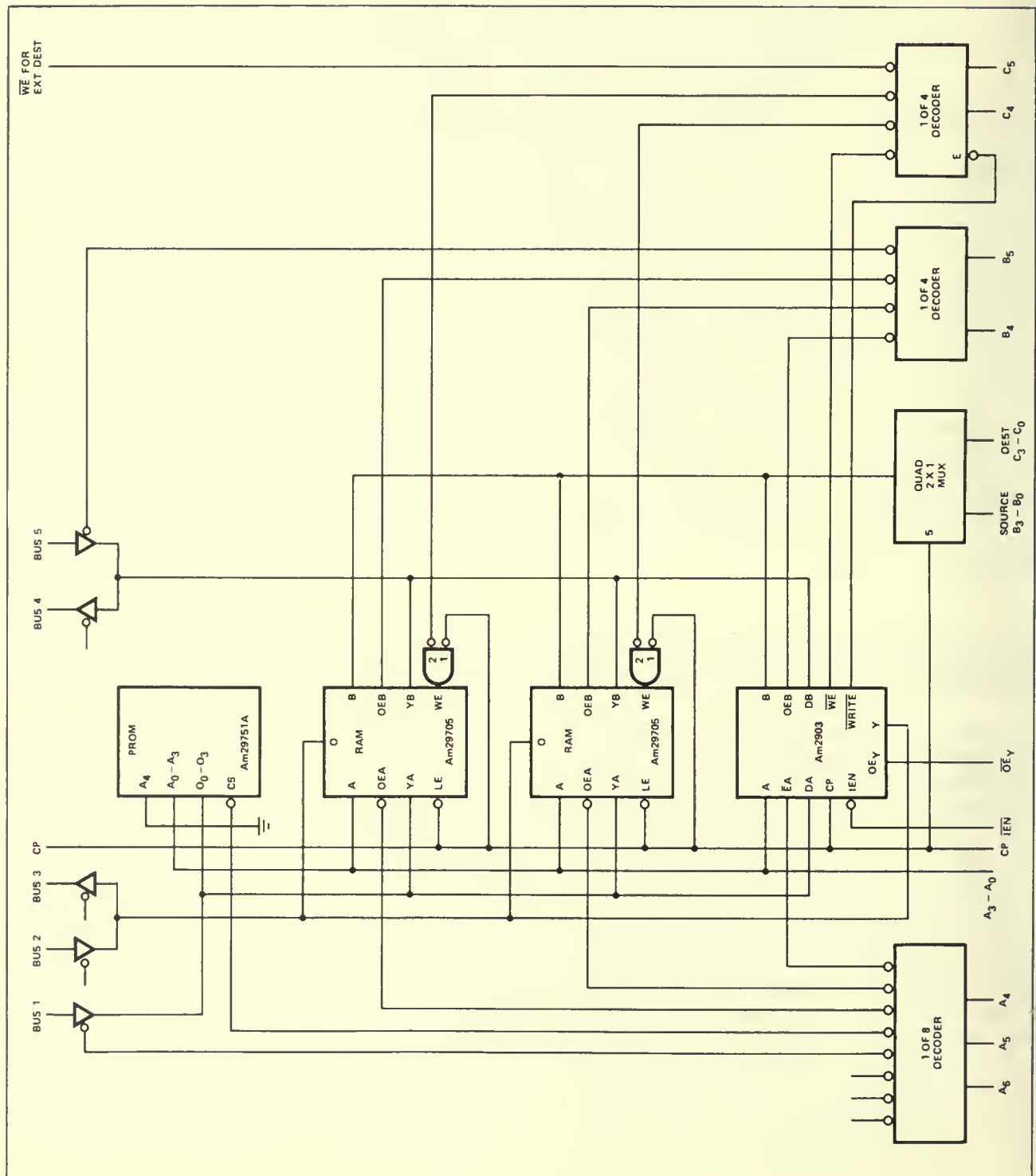


Fig. 26. Expanded memory.

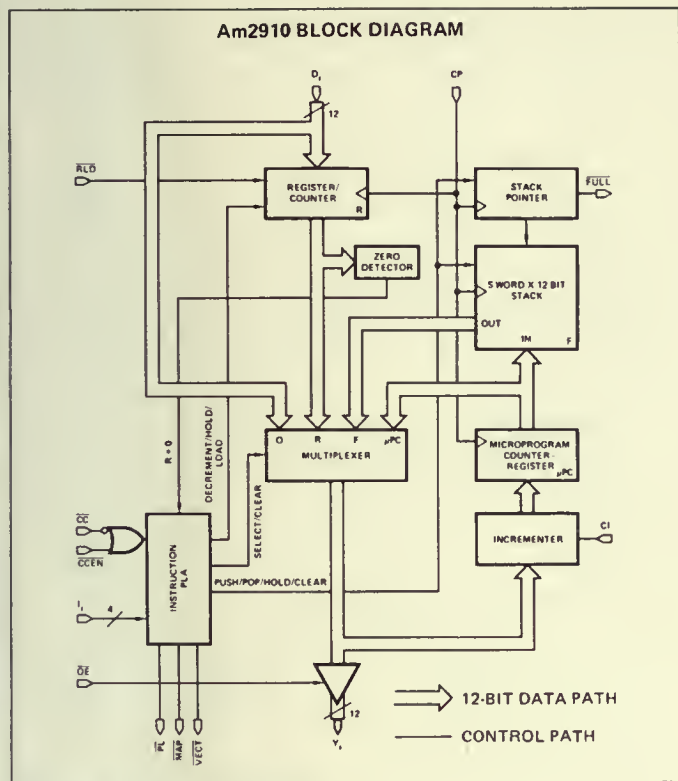


Fig. 27. Am2910 block diagram.

Architecture of the Am2910

The Am2910 is a bipolar microprogram controller intended for use in high-speed microprocessor applications. It allows addressing of up to 4096 words of microprogram. A block diagram of the Am2910 is shown in Fig. 27, and its application in a microcomputer is depicted in Fig. 28.

The controller contains a four-input multiplexer that is used to select either the register/counter, direct input, microprogram counter, or stack as the source of the next microinstruction address.

The register/counter consists of 12 *D*-type, edge-triggered flip-flops, with a common clock enable. When its load control, *RLD*, is LOW, new data is loaded on a positive clock transition. A few instructions include load; in most systems, these instructions will be sufficient, simplifying the microcode. The output of the register/counter is available to the multiplexer as a source for the next microinstruction address. The direct input furnishes a source of data for loading the register/counter.

The Am2910 contains a microprogram counter (μPC) that is

composed of a 12-bit incrementer followed by a 12-bit register. The μPC can be used in either of two ways: When the carry-in to the incrementer is HIGH, the microprogram register is loaded on the next clock cycle with the current *Y* output word plus one ($Y + 1 \rightarrow \mu PC$). Sequential microinstructions are thus executed. When the carry-in is LOW, the incrementer passes the *Y* output word unmodified so that μPC is reloaded with the same *Y* word on the next clock cycle ($Y \rightarrow \mu PC$). The same microinstruction is thus executed any number of times.

The third source for the multiplexer is the direct (*D*) input. This source is used for branching.

The fourth source available at the multiplexer input is a 5-word by 12-bit stack (file). The stack is used to provide return address linkage when executing microsubroutines or loops. The stack contains a built-in stack pointer (*SP*) which always points to the last file word written. This allows stack reference operations (looping) to be performed without a pop.

The stack pointer operates as an up/down counter. During microinstructions 1, 4, and 5, the PUSH operation is performed. This causes the stack pointer to increment and the file to be written with the required return linkage. On the cycle following the PUSH, the return data is at the new location pointed to by the stack pointer.

During five microinstructions, a POP operation may occur. The stack pointer decrements at the next rising clock edge following a POP, effectively removing old information from the top of the stack.

The stack pointer linkage is such that any sequence of pushes, pops, or stack references can be achieved. At RESET (instruction 0), the depth of nesting becomes 0. For each PUSH, the nesting depth increases by 1; for each POP, the depth increases by 1. The depth can grow to 5. After a depth of 5 is reached, *FULL* goes LOW. Any further PUSHes onto a full stack overwrite information at the top of the stack but leave the stack pointer unchanged. This operation will usually destroy useful information and is normally avoided. A POP from an empty stack may place non-meaningful data on the *Y* outputs but is otherwise safe. The stack pointer remains at 0 whenever a POP is attempted from a stack already empty.

The register/counter is operated during three microinstructions (8, 9, and 15) as a 12-bit down-counter, with result = zero available as a microinstruction branch test criterion. This provides efficient iteration of microinstructions. The register/counter is arranged so that if it is preloaded with a number *n* and then used as a loop termination counter, the sequence will be executed exactly *n* + 1 times. During instruction 15, a three-way branch under combined control of the loop counter and the condition code is available.

The device provides three-state *Y* outputs. These can be particularly useful in designs requiring automatic checkout of the

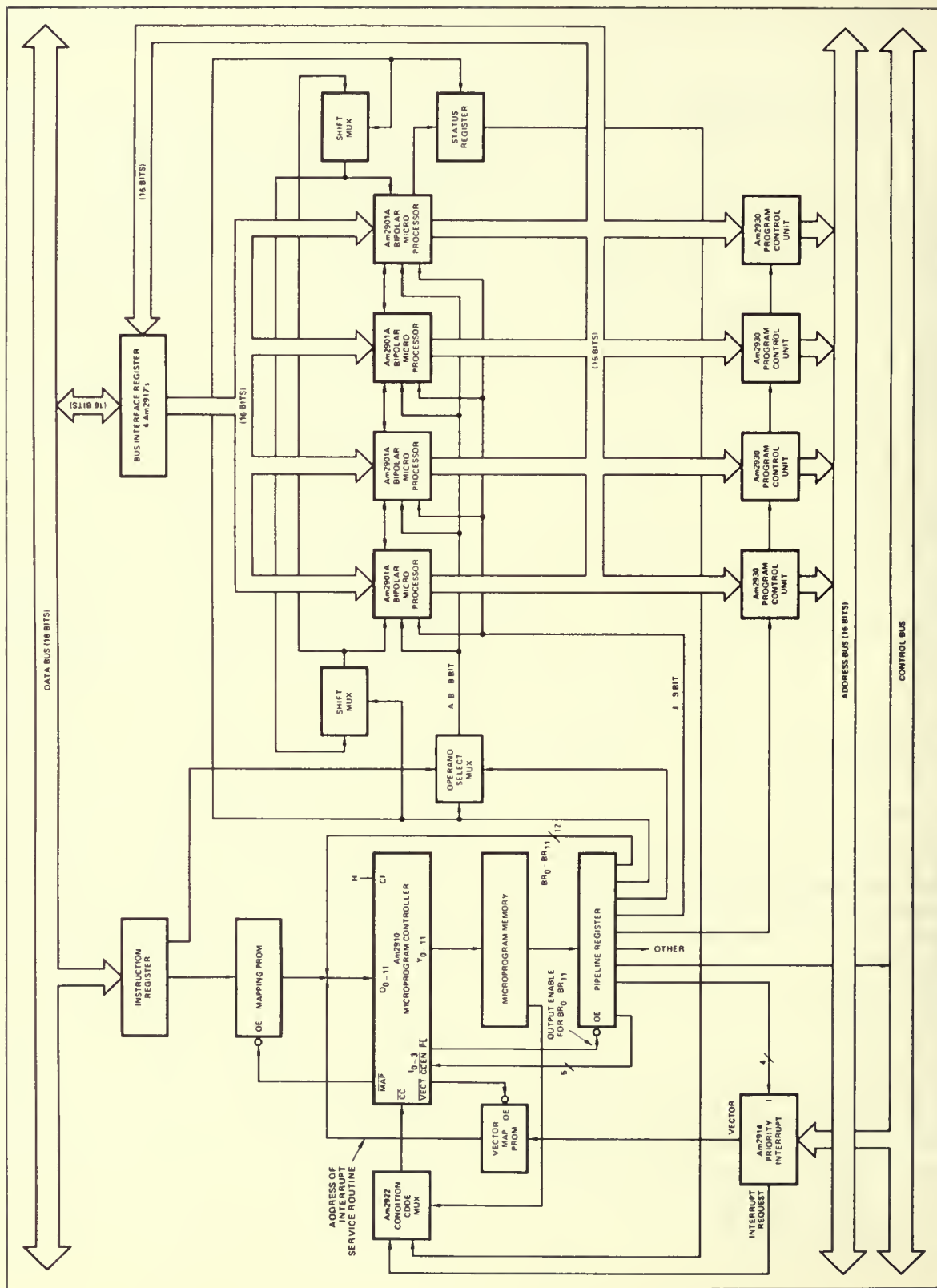


Fig. 28. Typical bipolar microcomputer using Am2910.

processor. The microprogram controller outputs can be forced into the high-impedance state, and pre-programmed sequences of microinstructions can be executed via external access to the address lines.

Operation

Table 6 shows the result of each instruction in controlling the multiplexer which determines the Y outputs, and in controlling the three enable signals $\overline{\text{PL}}$, $\overline{\text{MAP}}$, and $\overline{\text{VECT}}$. The effect on the register/counter and the stack after the next positive-going clock edge is also shown. The multiplexer determines which internal source drives the Y outputs. The value loaded into μPC is either identical to the Y output or else 1 greater, as determined by CI. For each instruction, one and only one of the three outputs $\overline{\text{PL}}$, $\overline{\text{MAP}}$, and $\overline{\text{VECT}}$ is LOW. If these outputs control three-state enables for the primary source of microprogram jumps (usually part of a pipeline register), a PROM which maps the instruction to a microinstruction starting location, and an optional third source (often a vector from a DMA or interrupt source), respectively, the three-state sources can drive the D inputs without further logic.

Several inputs, as shown in Table 7, can modify instruction execution. The combination CC HIGH and CCEN LOW is used as a test in 10 of the 16 instructions. $\overline{\text{RLD}}$, when LOW, causes the D input to be loaded into the register/counter, overriding any HOLD or DEC operation specified in the instruction. OE, normally LOW, may be forced HIGH to remove the Am2910 Y outputs from a three-state bus.

The Am2910 Instruction Set

The Am2910 provides 16 instructions which select the address of the next microinstruction to be executed. Four of the instructions are unconditional—their effect depends only on the instruction. Ten of the instructions have an effect which is partially controlled by an external, data-dependent condition. Three of the instructions have an effect which is partially controlled by the contents of the internal register/counter. The instruction set is shown in Table 6. In this discussion it is assumed that C_n is tied HIGH.

In the 10 conditional instructions, the result of the data-dependent test is applied to $\overline{\text{CC}}$. If the $\overline{\text{CC}}$ input is LOW, the test is considered to have been passed, and the action specified in the name occurs; otherwise, the test has failed and an alternate (often simply the execution of the next sequential microinstruction) occurs. Testing of $\overline{\text{CC}}$ may be disabled for a specific microinstruction by setting $\overline{\text{CCEN}}$ HIGH, which unconditionally forces the action specified in the name; that is, it forces a pass. Other ways of using $\overline{\text{CCEN}}$ include (1) tying it HIGH, which is useful if no microinstruction is data-dependent; (2) tying it LOW if data-

dependent instructions are never forced unconditionally; or (3) tying it to the source of Am2910 instruction bit I_0 , which leaves instructions 4, 6, and 10 as data-dependent but makes others unconditional. All of these tricks save one bit of microcode width.

The effect of three instructions depends on the contents of the register/counter. Unless the counter holds a value of zero, it is decremented; if it does hold zero, it is held and a different microprogram next address is selected. These instructions are useful for executing a microinstruction loop a known number of times. Instruction 15 is affected both by the external condition code and the internal register/counter.

Perhaps the best technique for understanding the Am2910 is to simply take each instruction and review its operation. In order to provide some feel for the actual execution of these instructions, Fig. 29 is included and depicts examples of all 16 instructions.

The examples given in Fig. 29 should be interpreted in the following manner: The intent is to show microprogram flow as various microprogram memory words are executed. For example, the CONTINUE instruction, instruction 14, as shown in Fig. 29, simply means that the contents of microprogram memory word 50 are executed and then the contents of word 51 are executed. This is followed by the contents of microprogram memory word 52 and the contents of microprogram memory word 53. The microprogram addresses used in the examples were arbitrarily chosen and have no meaning other than to show instruction flow. The exception to this is the first example, JUMP ZERO, which forces the microprogram location counter to address ZERO. Each dot refers to the time that the contents of the microprogram memory word is in the pipeline register. While no special symbology is used for the conditional instructions, the test to follow will explain what the conditional choices are in each example.

It might be appropriate at this time to mention that AMD has a microprogram assembler called AMDASM, which has the capability of using the Am2910 instructions in symbolic representation. AMDASM's Am2910 instruction symbolics (or mnemonics) are given in Fig. 29 for each instruction and are also shown in Table 6.

Instruction 0. JZ (JUMP and ZERO, or RESET) unconditional—specifies that the address of the next microinstruction is zero. Many designs use this feature for power-up sequences and provide the power-up firmware beginning at microprogram memory word location 0.

Instruction 1 is a CONDITIONAL JUMP-TO-SUBROUTINE via the address provided in the pipeline register. As shown in Fig. 29, the machine might have executed words at addresses 50, 51, and 52. When the contents of address 52 are in the pipeline register, the next address control function is the CONDITIONAL JUMP-TO-SUBROUTINE. Here, if the test is passed, the next instruction executed will be the contents of microprogram memory location 90. If the test has failed, the JUMP-TO-

Table 6 Instructions

Hex $I_8 I_7 I_6$	Mnemonic	Name	Reg/ cntr con- tents	Fail $\overline{CCEN} \neq \text{LOW and } \overline{CC} \neq \text{HIGH}$		PASS $\overline{CCEN} = \text{High or } \overline{CC} = \text{low}$		Reg/ cntr	Enable
				Y	STACK	Y	STACK		
0	JZ	JUMP ZERO	X	0	CLEAR	0	CLEAR	HOLD	PL
1	CJS	COND JSH PL	X	PC	HOLD	D	PUSH	HOLD	PL
2	JMAP	JUMP MAP	X	D	HOLD	D	HOLD	HOLD	MAP
3	CJP	COND JUMP PL	X	PC	HOLD	D	HOLD	HOLD	PL
4	PUSH	PUSH COND LD CNTR	X	PC	PUSH	PC	PUSH	†	PL
5	JSRP	COND JSB R/PL	X	R	PUSH	D	PUSH	HOLD	PL
6	CJV	COND JUMP VECTOR	X	PC	HOLD	D	HOLD	HOLD	VECT
7	JRP	COND JUMP R/PL	X	R	HOLD	D	HOLD	HOLD	PL
8	RFCT	REPEAT LOOP, CNTR $\neq 0$	$\neq 0$	F	HOLD	F	HOLD	DEC	PL
			$= 0$	PC	POP	PC	POP	HOLD	PL
9	RPCT	REPEAT LOOP, CNTR $\neq 0$	$\neq 0$	D	HOLD	D	HOLD	DEC	PL
			$= 0$	PC	HOLD	PC	HOLD	HOLD	PL
A	CRTN	COND RTN	X	PC	HOLD	F	POP	HOLD	PL
B	CJPP	COND JUMP PL & POP	X	PC	HOLD	D	POP	HOLD	PL
C	LDCT	LD CNTR & CONTINUE	X	PC	HOLD	PC	HOLD	LOAD	PL
D	LOOP	TEST END LOOP	X	F	HOLD	PC	POP	HOLD	PL
E	CONT	CONTINUE	X	PC	HOLD	PC	HOLD	HOLD	PL
F	TWB	THREE-WAY BRANCH	$\neq 0$	F	HOLD	PC	POP	DEC	PL
			$= 0$	D	POP	PC	POP	HOLD	PL

†If $\overline{CCEN} = \text{LOW}$ and $\overline{CC} = \text{HIGH}$, hold; else load. X = Don't Care

Table 7 Pin Functions

Abbreviation	Name	Function
D_i	Direct Input Bit i	Direct input to register/counter and multiplexer. D_0 is LSB.
I_i	Instruction Bit i	Selects one-of-sixteen instructions for the AM 2910.
\overline{CC}	Condition Code	Used as test criterion. Pass test is a LOW on \overline{CC} .
\overline{CCEN}	Condition Code Enable	Whenever the signal is HIGH, \overline{CC} is ignored and the part operates as though \overline{CC} were true (LOW).
CI	Carry-In	Low order carry input to incrementer for microprogram counter.
\overline{RLD}	Register Load	When LOW forces loading of register/counter regardless of instruction or condition.
\overline{OE}	Output Enable	Three-state control of Y_i outputs.
CP	Clock Pulse	Triggers all internal state changes at LOW-to-HIGH edge.
V_{cc}	+5 Volts	
GND	Ground	
Y_i	Microprogram Address Bit i	Address to microprogram memory. Y_0 is LSB, Y_{11} is MSB.
\overline{FULL}	FULL	Indicates that five items are on the stack.
\overline{PL}	Pipeline Address Enable	Can select #1 source (usually Pipeline Register) as direct input source.
\overline{MAP}	Map Address Enable	Can select #2 source (usually Mapping PROM or PLA) as direct input source.
\overline{VECT}	Vector Address Enable	Can select #3 source (for example, Interrupt Starting Address) as direct input source.

SUBROUTINE will not be executed; the contents of microprogram memory location 53 will be executed instead. Thus, the CONDITIONAL JUMP-TO-SUBROUTINE instruction at location 52 will cause the instruction either in location 90 or in location 53 to be executed next. If the TEST input is such that location 90 is selected, value 53 will be pushed onto the internal stack. This provides the return linkage for the machine when the subroutine beginning at location 90 is completed. In this example, the subroutine was completed at location 93 and a RETURN-FROM-SUBROUTINE was found at location 93.

Instruction 2 is the JUMP MAP instruction. This is an unconditional instruction which causes the \overline{MAP} output to be enabled so that the next microinstruction location is determined by the address supplied via the mapping PROMs. Normally, the JUMP MAP instruction is used at the end of the instruction fetch sequence for the machine. In the example of Fig. 29, microinstructions at locations 50, 51, 52, and 53 might have been the fetch sequence, and at its completion at location 53, the jump map function would be contained in the pipeline register. This example shows the mapping PROM outputs to be 90; therefore, an unconditional jump to microprogram memory address 90 is performed.

Instruction 3, CONDITIONAL JUMP PIPELINE, derives its branch address from the pipeline register branch address value (BR_0 – BR_{11} in Fig. 28). This instruction provides a technique for

branching to various microprogram sequences depending upon the test condition inputs. Quite often, state machines are designed which simply execute tests on various inputs waiting for the condition to come true. When the true condition is reached, the machine then branches and executes a set of microinstructions to perform some function. This usually has the effect of resetting the input being tested until some point in the future. Figure 29 shows the conditional jump via the pipeline register address at location 52. When the contents of microprogram memory word 52 are in the pipeline register, the next address will be either location 53 or location 30 in this example. If the test is passed, the value currently in the pipeline register (3) will be selected. If the test fails, the next address selected will be contained in the microprogram counter, which in this example is 53.

Instruction 4 is the PUSH/CONDITIONAL LOAD COUNTER instruction and is used primarily for setting up loops in microprogram firmware. In Figure 29, when instruction 52 is in the pipeline register, a PUSH will be made onto the stack and the counter will be loaded on the basis of the condition. When a PUSH occurs, the value pushed is always the next sequential instruction address. In this case, the address is 53. If the test fails, the counter is not loaded; if it is passed, the counter is loaded with the value contained in the pipeline register branch address field. Thus, a single microinstruction can be used to set up a loop to be executed a specific number of times. Instruction 8 will describe how to use the pushed value and the register/counter for looping.

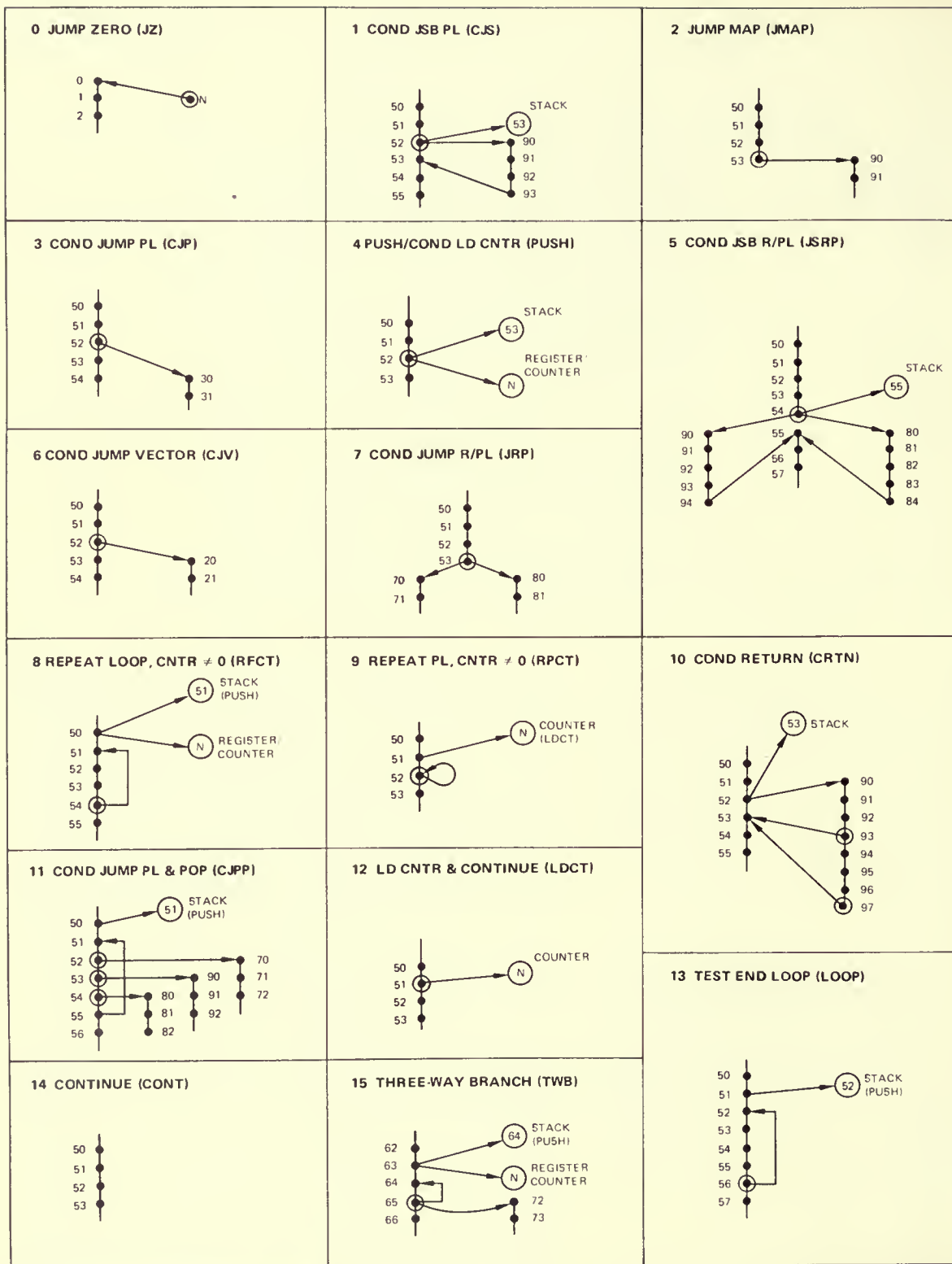


Fig. 29. Am2910 execution examples.

Instruction 5 is a **CONDITIONAL JUMP-TO-SUBROUTINE** via the register/counter or the contents of the Pipeline register. As shown in Fig. 29, a **PUSH** is always performed and one of two subroutines executed. In this example, either the subroutine beginning at address 80 or the subroutine beginning at address 90 will be performed. A return-from subroutine (instruction 10) returns the microprogram flow to address 55. In order for this microinstruction control sequence to operate correctly, both the next-address fields of instruction 53 and the next-address fields of instruction 54 have to contain the proper value. Let us assume that the branch address fields of instruction 53 contain the value 90 so that it will be in the Am2910 register/counter when the contents of address 54 are in the pipeline register. This requires that the instruction at address 53 load the register/counter. Now, during the execution of instruction 5 (at address 54), if the test fails, the contents of the register (value = 90) will select the address of the next microinstruction. If the test input passes, the pipeline register contents (value = 80) will determine the address of the next microinstruction. Therefore, this instruction provides the ability to select one of two subroutines to be executed based on a test condition.

Instruction 6 is a **CONDITIONAL JUMP VECTOR** instruction which provides the capability to take the branch address from a third source heretofore not discussed. In order for this instruction to be useful, the Am2910 output, **VECT**, is used to control a three-state control input of a register, buffer, or **PROM** containing the next microprogram address. This instruction provides one technique for performing interrupt-type branching at the microprogram level. Since this instruction is conditional, a pass causes the next address to be taken from the vector source, while failure causes the next address to be taken from the microprogram counter. In the example of Fig. 29, if the **CONDITIONAL JUMP VECTOR** instruction is contained at location 52, execution will continue at vector address 20 if the **TEST** input is **HIGH** and the microinstruction at address 53 will be executed if the **TEST** input is **LOW**.

Instruction 7 is a **CONDITIONAL JUMP** via the contents of the Am2910 register/counter or the contents of the pipeline register. This instruction is very similar to instruction 5, the **CONDITIONAL JUMP-TO-SUBROUTINE** via **R** or **PL**. The major difference between instruction 5 and instruction 7 is that no push onto the stack is performed with 7. Figure 29 depicts this instruction as a branch to one of two locations depending on the test condition. The example assumes the pipeline register contains the value 70 when the contents of address 52 are being executed. As the contents of address 53 are clocked into the pipeline register, the value 70 is loaded into the register/counter in the Am2910. The value 80 is available when the contents of address 53 are in

the pipeline register. Thus, control is transferred to either address 70 or address 80, depending on the test condition.

Instruction 8 is the **REPEAT LOOP, COUNTER \neq ZERO** instruction. This microinstruction makes use of the decrementing capability of the register/counter. To be useful, some previous instruction, such as 4, must have loaded a count value into the register/counter. This instruction checks to see whether the register/counter contains a non-zero value. If so, the register/counter is decremented, and the address of the next microinstruction is taken from the top of the stack. If the register/counter contains zero, the loop exit condition is occurring; control falls through to the next sequential microinstruction by selecting μPC ; the stack is **POPPed** by decrementing the stack pointer, but the contents of the top of the stack are thrown away.

An example of the **REPEAT LOOP, COUNTER \neq ZERO** instruction is shown in Fig. 29. In this example, location 50 most likely would contain a **PUSH/CONDITIONAL LOAD COUNTER** instruction which would have caused address 51 to be **PUSHed** onto the stack and the counter to be loaded with the proper value for looping the desired number of times.

In this example, since the loop test is made at the end of the instructions to be repeated (microaddress 54), the proper value to be loaded by the instructions at address 50 is one less than the desired number of passes through the loop. This method allows a loop to be executed 1 to 4096 times. If it is desired to execute the loop from 0 to 4095 times, the firmware should be written to make the loop exit test immediately after loop entry.

Single-microinstruction loops provide a highly efficient capability for executing a specific microinstruction a fixed number of times. Examples include fixed rotates, byte swap, fixed-point multiply, and fixed-point divide.

Instruction 9 is the **REPEAT PIPELINE REGISTER, COUNTER \neq ZERO** instruction. This instruction is similar to instruction 8 except that the branch address now comes from the pipeline register rather than the file. In some cases, this instruction may be thought of as a one-word file extension; that is, by using this instruction, a loop with the counter can still be performed when subroutines are nested five deep. This instruction's operation is very similar to that of instruction 8. The differences are that on this instruction, a failed test condition causes the source of the next microinstruction address to be the **D** inputs; and, when the test condition is passed, this instruction does not perform a **POP** because the stack is not being used.

In the example of Fig. 29, the **REPEAT PIPELINE, COUNTER \neq ZERO** instruction is instruction 52 and is shown as a single microinstruction loop. The address in the pipeline register would be 52. Instruction 51 in this example could be the **LOAD COUNTER AND CONTINUE** instruction (instruction 12). While

the example shows a single microinstruction loop, by simply changing the address in a pipeline register, multi-instruction loops can be performed in this manner for a fixed number of times as determined by the counter.

Instruction 10 is the conditional RETURN-FROM-SUBROUTINE instruction. As the name implies, this instruction is used to branch from the subroutine back to the next microinstruction address following the subroutine call. Since this instruction is conditional, the return is performed only if the test is passed. If the test is failed, the next sequential microinstruction is performed. The example in Fig. 29 depicts the use of the conditional RETURN-FROM-SUBROUTINE instruction in both the conditional and the unconditional modes. This example first shows a JUMP-TO-SUBROUTINE at instruction location 52, where control is transferred to location 90. At location 93, a conditional RETURN-FROM-SUBROUTINE instruction is performed. If the test is passed, the stack is accessed and the program will transfer to the next instruction at address 53. If the test is failed, the next microinstruction at address 94 will be executed. The program will continue to address 97, where the subroutine is complete. To perform an unconditional RETURN-FROM-SUBROUTINE, the conditional RETURN-FROM-SUBROUTINE instruction is executed unconditionally; the microinstruction at address 97 is programmed to force $\overline{\text{CCEN}}$ HIGH, disabling the test, and the forced PASS causes an unconditional return.

Instruction 11 is the CONDITIONAL JUMP PIPELINE register address and POP stack instruction. This instruction provides another technique for loop termination and stack maintenance. The example in Fig. 29 shows a loop being performed from address 55 back to address 51. The instructions at locations 52, 53, and 54 are all conditional JUMP and POP instructions. At address 52, if the TEST input is passed, a branch will be made to address 70 and the stack will be properly maintained via a POP. Should the test fail, the instruction at location 53 (the next sequential instruction) will be executed. Likewise, at address 53, either the instruction at 90 or 54 will be subsequently executed, depending on whether the test has been passed or failed. The instruction at 54 follows the same rules, going to either 80 or 55. An instruction sequence as described here, using the CONDITIONAL JUMP PIPELINE and POP instruction, is very useful when several inputs are being tested and the microprogram is looping waiting for any of the inputs being tested to occur before proceeding to another sequence of instructions. This provides the powerful jump-table programming technique at the firmware level.

Instruction 12 is the LOAD COUNTER AND CONTINUE instruction, which simply enables the counter to be loaded with the value at its parallel inputs. These inputs are normally

connected to the pipeline branch address field which (in the architecture being described here) serves to supply either a branch address or a counter value, depending upon whether the microinstruction has been executed. There are altogether three ways of loading the counter: the explicit load by this instruction I2, the conditional load included as part of instruction 4, and the use of the $\overline{\text{RLD}}$ input along with any instruction. The use of $\overline{\text{RLD}}$ with any instruction overrides any counting or decrementation specified in the instruction, calling for a load instead. Its use provides additional microinstruction power, at the expense of one bit of microinstruction width. This instruction I2 is exactly equivalent to the combination of instruction I4 and $\overline{\text{RLD}}$ LOW. Its purpose is to provide a simple capability to load the register/counter in those implementations which do not provide microprogrammed control for $\overline{\text{RLD}}$.

Instruction 13 is the TEST END-OF-LOOP instruction, which provides the capability of conditionally exiting a loop at the bottom; that is, this is a conditional instruction that will cause the microprogram to loop, via the file, if the test is failed or else to continue to the next sequential instruction. The example in Fig. 29 shows the TEST END-OF-LOOP microinstruction at address 56. If the test fails, the microprogram will branch to address 52. Address 52 is on the stack because a PUSH instruction has been executed at address 51. If the test is passed at instruction 56, the loop is terminated and the next sequential microinstruction at address 57 is executed, which also causes the stack to be POPped, thus accomplishing the required stack maintenance.

Instruction 14 is the CONTINUE instruction, which simply causes the microprogram counter to increment so that the next sequential microinstruction is executed. This is the simplest microinstruction of all and should be the default instruction which the firmware requests whenever there is nothing better to do.

Instruction 15, THREE-WAY BRANCH, is the most complex. It provides for testing of both a data-dependent condition and the counter during one microinstruction and provides for selecting among one of three microinstruction addresses as the next microinstruction to be performed. Like instruction 8, a previous instruction will have loaded a count into the register/counter while pushing a microbranch address onto the stack. Instruction 15 performs a decrement-and-branch-until-zero function similar to instruction 8. The next address is taken from the top of the stack until the count reaches zero; then the next address comes from the pipeline register. The above action continues as long as the test condition fails. If at any execution of instruction 15 the test condition is passed, no branch is taken; the microprogram counter register furnishes the next address. When the loop is ended, either because the count has become zero or because the

conditional test has been passed, the stack is POPped by decrementing the stack pointer, since interest in the value contained at the top of the stack is then complete.

The application of instruction 15 can enhance performance of a variety of machine-level instructions, for instance: (1) a memory search instruction to be terminated either by finding a desired memory content or by reaching the search limit, (2) variable-field-length arithmetic terminated early upon finding that the content of the portion of the field still unprocessed is all zeros, (3) key search in a disc controller processing variable-length records, and (4) normalization of a floating-point number.

As one example, consider the case of a memory search instruction. As shown in Fig. 29, the instruction at microprogram address 63 can be instruction 4 (PUSH), which will push the value 64 onto the microprogram stack and load the number n , which is

one less than the number of memory locations to be searched before giving up. Location 64 contains a microinstruction which fetches the next operand from the memory area to be searched and compares it with the search key. Location 65 contains a microinstruction which tests the result of the comparison and also is a THREE-WAY BRANCH for microprogram control. If no match is found, the test fails and the microprogram goes back to location 64 for the next operand address. When the count becomes zero, the microprogram branches to location 72, which does whatever is necessary if no match is found. If a match occurs on any execution of the THREE-WAY BRANCH at location 65, control falls through to location 66, which handles this case. Whether the instruction ends by finding a match or not, the stack will have been POPped once, removing the value 64 from the top of the stack.

```

AM2903 :=
begin
! ISPS description of the AM2903 4 bit slice microprocessor.

! Page 1 of the description contains declarations of simple carriers
! **PC.State** and **MP.SIAIF** sections describe the actual
! carriers contained within the AM2903 chip.
! **External.State** describes the simple carriers that
! terminate in pins.
! **Implementation.Declarations** describe carriers necessary
! for the ISP description.

! Page 2 describes the access computations used to source and sink
! computation data.

! Page 3 contains descriptions of the basic operation cycle and
! the actual instruction execution.

! Page 5 - 9 contain descriptions of computations for the 2, G.W,
! P.OVR, and Cn4 output pins plus the "gi" and "pi"
! intermediate carry generate and carry propagate computations.

**PC.State**
    ASHF<3:0>,      ! ALU shifter
    R<3:0>,          ! R inputs to ALU
    S<3:0>,          ! S inputs to ALU

**MP.State**
    RAM[0:15]<3:0>, ! 16 X 4 bit 2 port RAM

**External.State**
    A<3:0>,          ! A RAM port input address
    B<3:0>,          ! B RAM port input address
    Cn<>,            ! Carry in
    DA<3:0>,         ! Direct data input (R input)
    DB<3:0>,         ! Direct data input (S input)
    EA<>,            ! HIGH => R = DA, LOW => R = A
    F<3:0>,          ! Sign of ALU (MSS)
    IEN<>,           ! Output from ALU
    LSSL<>,          ! Instruction enable (LOW true)
    OEB<>,          ! LOW => least Significant Slice
    OEW<>,          ! LOW enables RAM port B
    OEY<>,          ! LOW enables ALU shift to Y
    Q<3:0>,          ! Output from Q register
    Q103<>,         ! Q register shift MSB
    Q100<>,         ! Q register shift LSB
    S103<>,         ! ALU shift MSB
    S100<>,         ! ALU shift LSB
    W.MSS<>,        ! LSS=LOW => NOI WRITE output
    WE<>,           ! ISS=HIGH => input pin:
    Y<3:0>,         ! HIGH => IS, LOW => MSS
    I<0:0>,         ! Write Enable: LOW => RAM = Y
    I<0:0>,         ! Data input/output
    I<0:0>,         ! Instruction inputs

**Implementation.Declarations**
    p<>,            ! Accumulator for computed P
    g<>,            ! Accumulator for computed G
    cn3<>,          ! Accumulator for computed Cn+3
    write<>,        ! Internal write flag

    dest<3:0> := I<8:5>, ! Destination select
    op<3:0> := I<4:1>, ! Function OP code
    iq<> := I<0>, ! I<0> (part of source select)

    macro hiz := '1111', ! Tristate constant
    macro parity := I<3> xor F<2> xor F<1> xor F<0> xor S103,
    macro mss := I<SSL and (not W.MSS)].

**Access.Computation**{us}
    source :=
    begin
        DECODE EA @ I<0> @ OEB =>
        begin
            #0 := (R = RAM[A]; S = RAM[D]),
            #1 := (R = ram[a]; s = db ),
            #2:#3 := (H = RAM[A]; S = Q ),
            #4 := (R = DA : S = RAM[H]),
            #5 := (R = DA : S = UB ),
            #6:#7 := (R = DA : S = Q )
        end
    end,

    destination :=
    begin
        DECODE I<8:7> =>
        begin
            D := begin
                ! ASHFT = F/2
                ASHF @ S100 = S103 @ F next
                IF mss and (not I<5>) => ASHF<3>@ASHF<2> = ASHF<2>@ASHF<3>;
                WRITE = 0;
                DECODE I<8> =>
                begin
                    0 := q103 @ q100 = hiz,
                    1 := q @ q100 = q103 @ Q
                end
                and,
                1 := begin
                    ! ASHF = F
                    ASHF = F; S100 = parity; WRITE = I<0> xor I<5>;
                    DECODE I<6:5> =>
                    begin
                        0 := Q103 = Q100 = hiz,
                        1 := 0 @ Q100 = Q103 @ Q,
                        2:3 := (Q103 = Q100 = hiz; Q = F)
                    end
                    and,
                    2 := begin
                        ! ASHF = 2F
                        S103 @ ASHF = F @ S100 next
                        IF mss and (not I<5>) => S103 @ ASHF<3> = ASHF<3> @ S103;
                        WRITE = 0;
                        DECODE I<4> =>
                        begin
                            0 := Q103 = Q100 = hiz,
                            1 := Q103 @ Q = Q @ Q100
                        end
                        and,
                        3 := begin
                            ! ASHF = <F>, <S100>
                            DECODE I<0:5> eq1 '10 =>
                            begin
                                0 := (ASHF = F; S103 = F<3>; S100 = hiz),
                                1 := ASHF <= S100
                            end
                            and,
                            WRITE = not I<0>;
                            DECODE I<8:5> eq1 '01 =>
                            begin
                                0 := Q103 = Q100 = hiz,
                                1 := Q103 @ Q = Q @ Q100
                            end
                        end
                    end
                end
            end
        end
    end,

    run :=
    begin
        WRITE = 1 next
        DECODE IEN =>
        begin
            0 := begin
                source() next
                exec() next
                IF I<4:0> NEQ 0 => destination() next
                IF not LSSL => W.MSS = WRITE next
                IF not OEY => Y = ASHF next
                IF not WE => RAM[B] = Y next
                Z() next
                g(): pi() next
                Cn4(): P.OVR(): G.W()
            end
            1 := WRITE = 1
        end next
        RESTART run
    end,

    **Instruction.Cycle**
    ! Main instruction cycle

    **Instruction.Execution**{us}
    exec :=
    begin
        DECODE I<4:1> =>
        begin
            "0 := (DECODE I<0> =>
            begin
                0 := (special.functions()),
                1 := F = '1111
            end),
            "1 := F = ((S - R) - 1) + Cn,
            "2 := F = ((R - S) - 1) + Cn,
            "3 := F = (R + S) + Cn,
            "4 := F = S + Cn,
            "5 := F = (not S) + Cn,
            "6 := F = R + Cn,
            "7 := F = (not R) + Cn,
            "8 := F = 0,
            "9 := F = (not R) and S,
            "A := F = R eqv S,
            "B := F = R xor S,
            "C := F = R and S,
            "D := F = not (R or S),
            "E := F = not (R and S),
            "F := F = R or S,
        end
    end,

```

! Special functions are decoded from I<B:S> when I<4:0> equal zero.
! These are the built-in multiplication, division, and normalization
! functions.

special.functions :=

begin
DECODE I<B:S> =>

begin

"0:"3 := begin

DECODE Z =>

begin

0 := F = S + Cn.

1 := F = (S + R) + Cn

end next

DECODE mss =>

begin

0 := (ASHFT @ SIO0 = SIO3 @ F; WRITE = 0).

1 := (SIO3 = hiz;

DECODE I =>

begin

0 := ASHFT @ SIO0 = Cn4 @ F.

1 := ASHFT @ SIO0 = (F<3> xor P.OVR()) @ F

end)

end

"4:"S := begin

DECODE I<S> =>

begin

0 := ASHFT = F = (S + 1) + Cn.

1 := (DECODE Z =>

begin

0 := ASHFT = F = S + Cn.

1 := ASHFT = F = (not S) + Cn

end next

If mss => ASHFT<3> = S<3> xor F<3>

end;

SIO0 = parity; QIO3 = QIO0 = hiz; WRITE = 0

end.

"6:"7 := begin

DECODE Z =>

begin

0 := F = S + Cn.

1 := F = (not S) + Cn

end next

DECODE mss =>

begin

0 := ASHFT @ SIO0 = SIO3 @ F.

1 := (SIO3 = hiz;

ASHFT @ SIO0 = (F<3> xor P.OVR()) @ F)

end;

Q @ QIO0 = QIO3 @ Q; WRITE = 0

and,

"8:"0 := begin

f = S + Cn next

DECODE I =>

begin

0 := (ASHFT = f; SIO3 = F<3>; SIO0 = hiz).

1 := (DECODE mss =>

begin

0 := SIO3 @ ASHFT = f @ SIO0.

1 := SIO3 @ ASHFT

= (R<3> xor F<3>) @ F<2:0> @ SIO0

end)

end;

QIO3 R Q = Q @ QIO0; WRITE = 0

and,

"C:"F := begin

DECODE Z =>

begin

0 := F = (S + R) + Cn.

1 := F = ((S - R) - 1) + Cn

end next

DECODE I =>

begin

0 := DECODE mss =>

begin

0 := SIO3 @ ASHFT = F @ SIO0.

1 := SIO3 @ ASHFT

= (R<3> xor F<3>) @ F<2:0> @ SIO0

end.

1 := (ASHFT = F; SIO3 = F<3>; SIO0 = hiz)

end;

QIO3 @ Q = Q @ QIO0; WRITE = 0

end

and

end.

==Service.Facilities==(us]

Z()<> :=

begin

DECODE I<4:1> =>

begin

"0 := DECODE I<D> =>

begin

0 := DECODE I<B:6> =>

begin

["0:"3,"6:"7] := If not LSSL => Z = Q<D>.

"4 := Z = ASHFT eq1 0.

"S := If mss => Z = S<3>.

"8:"9 := Z = Q eq1 0.

"A:"B := Z = (Q eq1 0) and (F eq1 0).

"C:"F := If mss => Z = R<3> eqv F<3>

end.

1 := Z = ASHFT eq1 0

end.

"1:"F := Z = ASHFT eq1 0

end.

gi()<3:0> :=

begin

DECODE I<4:1> =>

begin

"0 := begin

DECODE I<D> =>

begin

0 := DECODE I<B:6> =>

begin

"0:"3 := DECODE Z =>

begin

0 := gi = 0.

1 := gi = R and S

end.

"4 := DECODE LSSL =>

begin

0 := gi = '000 @ S<D>.

1 := gi = 0

end.

["S,"8:"B] := gi = 0.

"8:"7 := DECODE Z =>

begin

0 := gi = 0.

1 := gi = (not R) and S

end.

"C:"F := DECODE Z =>

begin

0 := gi = R and S.

1 := gi = (not R) and S

and

end.

1 := gi = 0

end

["1,"9,"8] := gi = (not R) and S.

"2 := gi = R and (not S).

["3,"A,"C,"E] := gi = R and S.

"4:"8 := gi = 0.

["0,"F] := gi = (not R) and (not S)

end

end.

pi()<3:0> :=

begin

DECODE I<4:1> =>

begin

"0 := DECODE I<D> =>

begin

0 := DECODE I<B:6> =>

begin

"0:"3 := DECODE Z =>

begin

0 := pi = S.

1 := pi = R or S

end.

"4 := DECODE LSSL =>

begin

0 := pi = S<3:1> @ 1.

1 := pi = S

end.

"5 := DECODE Z =>

begin

0 := pi = S.

1 := pi = not S

and.

"8:"7 := DECODE Z =>

begin

0 := pi = S.

1 := pi = (not R) or S

end.

"8:"B := pi = S.

"C:"F := DECODE Z =>

begin

0 := pi = R or S.

1 := pi = (not R) or S

and


```

end,
1 := pi = '1111
and,
["1:"B]:= pi = (not R) or S,
"2 := pi = R or (not S),
["3:"A]:= pi = R or S,
"4 := pi = S,
"5 := pi = not S,
"6 := pi = R,
"7 := pi = not R,
["8:"G,"C:"F]:= pi = '1111
and
and,
! Carry Out
Cn4(<)> :=
begin
DECODE I<4:1> =>
begin
"D := DECODE I<D> =>
begin
D := DECODE I<8:5> =>
begin
["0:"7,"C:"F]:= Cn4 = g or (p and Cn),
"8:"9 := DECODE mss =>
begin
D := Cn4 = g or (p and Cn),
1 := Cn4 = Q<3> xor Q<2>
end,
"A:"B := DECODE mss =>
begin
D := Cn4 = g or (p and Cn),
1 := Cn4 = F<3> xor F<2>
end
and,
1 := Cn4 = D
end,
"1:"7 := Cn4 = g or (p and Cn),
"8:"F := Cn4 = D
end
and,
P.OVR(<)> := ! (not P)/OVR pin
begin
p = pi() eqv '1111 next
DECODE mss =>
begin
D := DECODE I<4:1> =>
begin
"D := P.OVR = (not p) and (not I<D>),
"1:"7 := P.OVR = not p,
"8:"F := P.OVR = D
end,
1 := DECODE I<4:1> =>
begin
"D := DECODE I<D> =>
begin
D := DECODE I<8:5> =>
begin
["0:"7,"C:"F]:= P.OVR = Cn3 xor Cn4,
"8:"9 := P.OVR = Q<2> xor Q<1>,
"A:"B := P.OVR = F<2> xor F<1>
end,
1 := P.OVR = D
end,
"1:"7 := P.OVR = Cn3 xor Cn4,
"8:"F := P.OVR = D
end
end
and
G.N(<)> := ! (not G)/N pin
begin
DECODE mss =>
begin
D := G.N = not g,
1 := DECODE I<4:1> =>
begin
"D := DECODE I<D> =>
begin
D := DECODE I<8:5> =>
begin
["0:"4,"6:"7,"A:"F]:= G.N = F<3>,
"5 := DECODE Z =>
begin
D := G.N = F<3>,
1 := G.N = F<3> xor s<3>
end,
"8:"9 := G.N = Q<3>
end,
1 := G.N = F<3>
end,
"1:"F := G.N = F<3>
end
end
end
end
! End of AM2903 description

```

```

AM2910 :=
begin
! ISPS description of AMD AM2910 microprogram sequencer.
! The AM2910 is a 12 bit microprogram address controller.
! The controller is designed to be used with the AM2901 or the AM2903
! microprocessor slice and external memory.
! Simulation of the AM2910 alone is possible, but emulation of any
! computer systems requires that this description be joined with
! the AM2901 or AM2903 description.
**PC.State**
uPC<11:0>, ! Microprogram counter
R<11:0>, ! Address register
SP<2:0>, ! Stack pointer
STACK[0:5]<11:0>, ! Stack register file
**External.State**
macro p1, := '011, ! Pipeline address enable
macro map, := '101, ! Map address enable
macro vect, := '110, ! Vector address enable
CI<>, ! Carry in
CC<>, ! Condition code enable input bit
CCEN<>, ! Direct inputs
O<11:0>, ! Stack full flag
FULL<>, ! Instruction register
ir<6:0>, ! Instruction vector affecting 2910
I<3:0> := ir<3:0>, ! Output enable control line
OE<>, ! Register load
RLDC<>, ! Map address enable flag
MAP<>, ! Pipeline address enable flag
PL<>, ! Vector address enable flag
VECT<>, := enable<0>,
**Implementation.Variables**
enable<2:0>, ! Enable conditions
fail<>, ! CC fail flag
pass<>, ! CC pass flag
macro h1z := 'FFFFF, ! High impedance constant
**Operation.Cycle**(<us>)
run(main) :=
begin
! Basic operation loop
IF not RLD => R = 0 next ! Forced (external) load of reg.
Y() next ! Put out selected address
uPC = Y + CT next ! Increment pc
RESTART run
end,
**Address.Source.Selection**(<us>)
Y(<11:0>) :=
begin
IF i eq1 "2 => enable = map;
IF i eq1 "6 => enable = vect.;
IF (i neq "2) and (i neq "6) => enable = p1;
fail = (not CCEN) and CC next
pass = not fail next
DECODE i =>
begin
"0 := J2 := (Y = SP = 0; FULL = 1),
"1 := CJS := (IF fail => {Y = uPC};
IF pass => {Y = D; push.()}),
"2 := JMAP := (Y = D),
"3 := CJP := (IF fail => {Y = uPC};
IF pass => {Y = D}),
"4 := PUSH := (Y = uPC; push.(); IF pass => R = 0),
"5 := JSRP := (IF fail => {Y = R};
IF pass => {Y = D; push.()}),
"6 := CJV := (IF fail => {Y = uPC};
IF pass => {Y = 0}),
"7 := JRP := (IF fail => {Y = R};
IF pass => {Y = D}),
"8 := RFCT := (IF R eq1 0 => {Y = uPC; pop()} next
IF R neq 0 => {Y = STACK[SP]; R = R - 1}),
"9 := RPCT := (IF R eq1 0 => {Y = uPC} next
IF R neq 0 => {Y = 0; R = R - 1}),
"A := CRIN := (IF fail => {Y = uPC};
IF pass => {Y = STACK[SP] next pop()}),
"B := CJPP := (IF fail => {Y = uPC};
IF pass => {Y = D; pop()}),
"C := LDCT := (Y = uPC; R = D),
"D := LOOP := (IF fail => {Y = STACK[SP]};
IF pass => {Y = uPC; pop()}),
"E := CONT := (Y = uPC),
"F := IWB := (DECODE R eq1 0 =>
begin
D := {IF fail => {Y = STACK[SP]};
IF pass => {Y = uPC; pop()}; R = R - 1},
1 := {IF fail => {Y = D};
IF pass => {Y = uPC; pop()}
end)
end next
IF OE => Y = h1z
end,
pop := (IF SP neq 0 => SP = SP - 1; FULL = 1),
push. :=
begin
DECODE SP eq1 #4 =>
begin
D := (FULL = 1; SP = SP + 1),
1 := (FULL = 0)
end next
STACK[SP] = uPC
end
end
! End of AM2910 description

```

APPENDIX 2 (right) AM2910 ISP DESCRIPTION