

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 to determine the address of the next microinstruction to be executed. That address ² is sent to the ROM, and the next microinstruction ³ 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 must occur during a single clock cycle. During the same time, the loop from ¹ to ⁴ 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 Am2DOO Family Data Book*, Advanced Micro Devices, Inc., 1976.

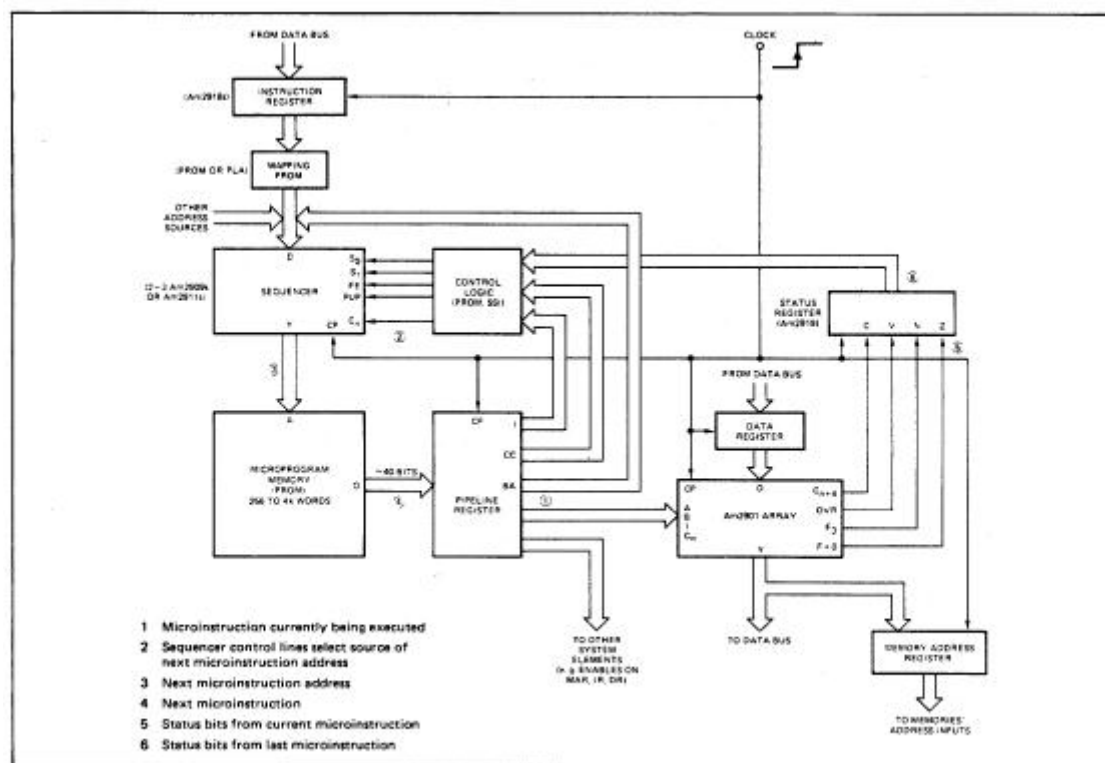


Fig. 1

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

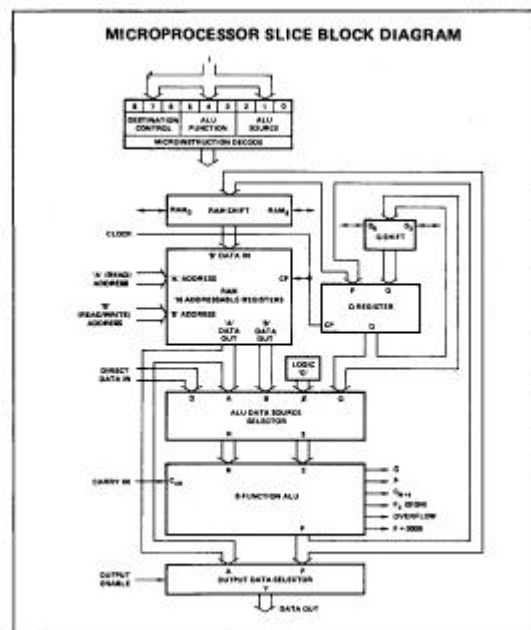


Fig. 2. Microprocessor slice block diagram.

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 I_0 , I_1 , and I_2 inputs. The definitions of I_0 , I_1 , and I_2 for the eight source operand combinations are as shown in Table 1. Also shown is the octal code for each selection.

The I_3 , I_4 , and I_5 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, G, and carry propagate, P, are outputs of the device for use with a carry-lookahead generator such as the

Table 1 ALU Source Operand Control

<i>Microcode</i>				<i>ALU source operands</i>	
I_2	I_1	I_0	<i>Octal code</i>	<i>R</i>	<i>S</i>
<i>L</i>	<i>L</i>	<i>L</i>	<i>0</i>	<i>A</i>	<i>Q</i>
<i>L</i>	<i>L</i>	<i>H</i>	<i>1</i>	<i>A</i>	<i>B</i>
<i>L</i>	<i>H</i>	<i>L</i>	<i>2</i>	<i>O</i>	<i>Q</i>

<i>L</i>	<i>H</i>	<i>H</i>	3	<i>O</i>	<i>B</i>
<i>H</i>	<i>L</i>	<i>L</i>	4	<i>O</i>	<i>A</i>
<i>H</i>	<i>L</i>	<i>H</i>	5	<i>D</i>	<i>A</i>
<i>H</i>	<i>H</i>	<i>L</i>	6	<i>D</i>	<i>Q</i>
<i>H</i>	<i>H</i>	<i>H</i>	7	<i>D</i>	<i>O</i>

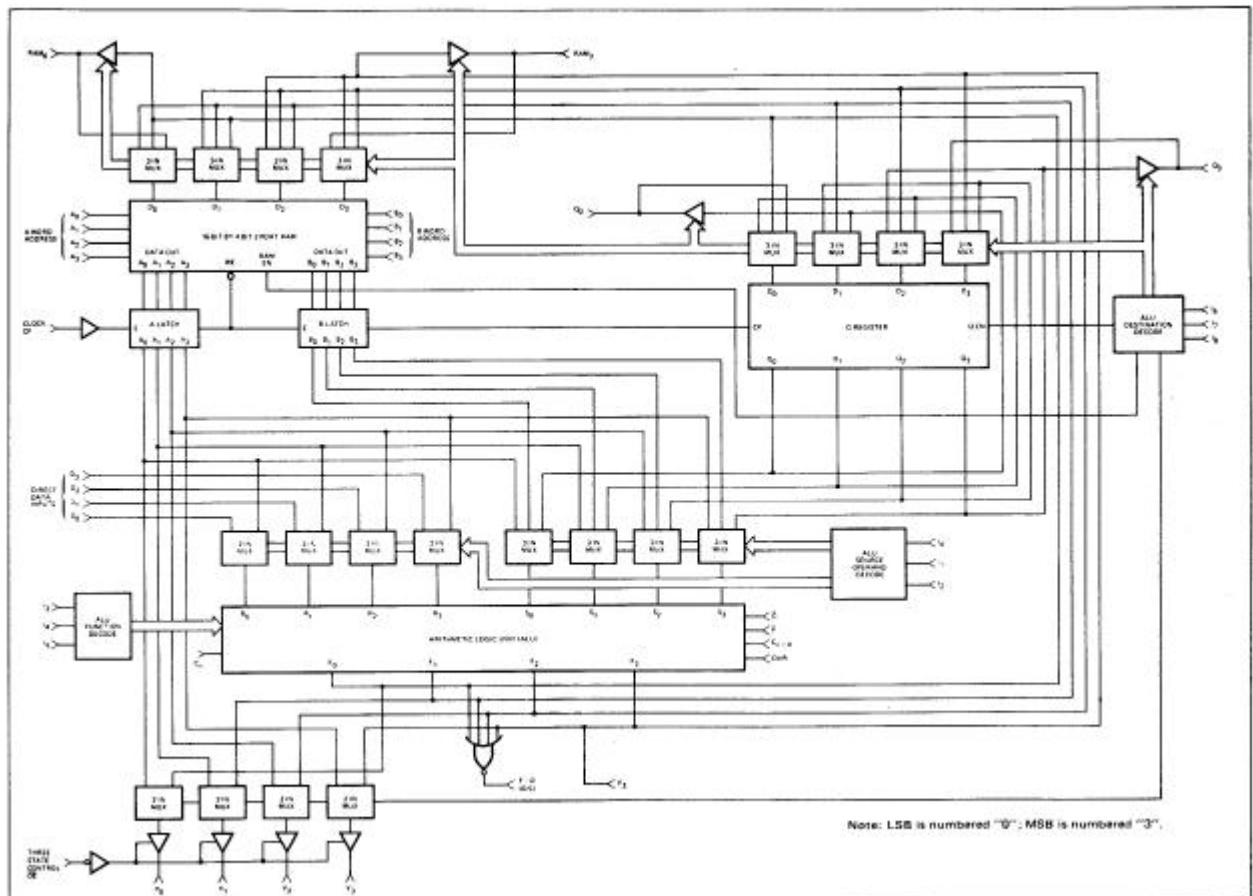


Fig. 3. Detailed Am2901 microprocessor block diagram.

<i>Microcode</i>				<i>ALU function</i>	<i>Symbol</i>
<i>I</i> ₅	<i>I</i> ₄	<i>I</i> ₃	<i>Octal code</i>		
<i>L</i>	<i>L</i>	<i>L</i>	0	R plus S	R+S
<i>L</i>	<i>L</i>	<i>H</i>	1	S minus R	S-R
<i>L</i>	<i>H</i>	<i>L</i>	2	R minus S	R-S
<i>L</i>	<i>H</i>	<i>H</i>	3	R OR S	R V S
<i>H</i>	<i>L</i>	<i>L</i>	4	R AND S	R ^ S

<i>H</i>	<i>L</i>	<i>H</i>	5	R AND S	$\overline{R \wedge S}$
<i>H</i>	<i>H</i>	<i>L</i>	6	R EX-OR S	$R \vee S$
<i>H</i>	<i>H</i>	<i>H</i>	7	R EX-NOR S	$\overline{R \vee S}$

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 (OE) is used to enable the three-state outputs. When 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 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.

Table 3 ALU Destination Control

<i>Microcode</i>				<i>RAM function</i>		<i>Q-register function</i>			<i>RAM shifter</i>		<i>Q shifter</i>	
<i>I₈</i>	<i>I₇</i>	<i>I₆</i>	<i>Octal Code</i>	<i>Shift</i>	<i>Load</i>	<i>Shift</i>	<i>Load</i>	<i>Y output</i>	<i>RAM₀</i>	<i>RAM₃</i>	<i>Q₀</i>	<i>Q₃</i>
<i>L</i>	<i>L</i>	<i>L</i>	<i>0</i>	<i>X</i>	None	None	F→ Q	<i>F</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>
<i>L</i>	<i>L</i>	<i>H</i>	<i>1</i>	<i>X</i>	None	<i>X</i>	None	<i>F</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>
<i>L</i>	<i>H</i>	<i>L</i>	<i>2</i>	None	F→ B	<i>X</i>	None	<i>A</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>
<i>L</i>	<i>H</i>	<i>H</i>	<i>3</i>	None	F→ B	<i>X</i>	None	<i>F</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>
<i>H</i>	<i>L</i>	<i>L</i>	<i>4</i>	Down	F/2→ B	Down	$Q/2 \rightarrow Q$	<i>F</i>	<i>F₀</i>	<i>IN₃</i>	<i>Q₀</i>	<i>IN₃</i>
<i>H</i>	<i>L</i>	<i>H</i>	<i>5</i>	Down	F/2→ B	<i>X</i>	None	<i>F</i>	<i>F₀</i>	<i>IN₃</i>	<i>Q₀</i>	<i>X</i>
<i>H</i>	<i>H</i>	<i>L</i>	<i>6</i>	Up	2F→ B	Up	2Q→ Q	<i>F</i>	<i>IN₀</i>	<i>F₃</i>	<i>IN₀</i>	<i>Q₃</i>
<i>H</i>	<i>H</i>	<i>H</i>	<i>7</i>	Up	2F→ B	<i>X</i>	None	<i>F</i>	<i>IN₀</i>	<i>F₃</i>	<i>X</i>	<i>Q₃</i>

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.

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 II and S inputs are the two inputs selected according to Table 1.

Table 4 Source Operand and ALU Function Matrix

Octal $I_{5,4,3}$	ALU function	Octal $I_{2,1,0}$ ALU source							
		0 A, Q	1 A, B	2 Q, Q	3 Q, B	4 Q, A	5 D, A	6 D, Q	7 D, Q
0	$C_n = L$ R plus S $C_n = H$	$A + Q$ $A + Q + 1$	$A + B$ $A + B + 1$	Q $Q + 1$	B $B + 1$	A $A + 1$	$D + A$ $D + A + 1$	$D + Q$ $D + Q + 1$	D $D + 1$
1	$C_n = L$ S minus R $C_n = H$	$Q - A - 1$ $Q - A$	$B - A - 1$ $B - A$	$Q - 1$ Q	$B - 1$ B	$A - 1$ A	$A - D - 1$ $A - D$	$Q - D - 1$ $Q - D$	$-D - 1$ $-D$
2	$C_n = L$ R minus S $C_n = H$	$A - Q - 1$ $A - Q$	$A - B - 1$ $A - B$	$-Q - 1$ $-Q$	$-B - 1$ $-B$	$-A - 1$ $-A$	$D - A - 1$ $D - A$	$D - Q - 1$ $D - Q$	$D - 1$ 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	$\bar{A} \nabla Q$	$\bar{A} \nabla B$	\bar{Q}	\bar{B}	\bar{A}	$\bar{D} \nabla A$	$\bar{D} \nabla Q$	\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	$\bar{A} \nabla Q$
7	1		$\bar{A} \nabla B$
7	5		$\bar{D} \nabla A$
7	6		$\bar{D} \nabla Q$
7	2	INVERT	\bar{Q}
7	3		\bar{B}
7	4		\bar{A}
7	7		\bar{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	$\bar{A} \wedge Q$
5	1		$\bar{A} \wedge B$
5	5		$\bar{D} \wedge A$
5	6		$\bar{D} \wedge Q$

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 RAM ₃ 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} .
OE	Output enable. When OE is HIGH, the Y outputs are OFF; when OE is LOW, the Y outputs are active (HIGH or LOW).
P, 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 $F_{0\sim3}$ 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 $I_{2,1,0}$	$C_n = 0$ (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)

$$\begin{aligned}
 P_0 &= R_0 + S_0 & G_0 &= R_0 S_0 & C_n &= G_2 + P_2 G_2 + P_2 P_2 G_1 + P_2 P_2 P_2 G_0 + P_2 P_2 P_2 C_n \\
 P_1 &= R_1 + S_1 & G_1 &= R_1 S_1 & C_2 &= G_2 + P_2 G_2 + P_2 P_2 G_0 + P_2 P_2 P_2 C_n \\
 P_2 &= R_2 + S_2 & G_2 &= R_2 S_2 & & \\
 P_3 &= R_3 + S_3 & G_3 &= R_3 S_3 & &
 \end{aligned}$$

$I_{2,1,0}$	Function	\bar{P}	\bar{G}	C_{n+1}	OVR
0	R + S	$\bar{P}_2 \bar{P}_1 \bar{P}_0$	$\bar{G}_3 + P_2 \bar{G}_2 + P_2 P_2 \bar{G}_1 + P_2 P_2 P_2 \bar{G}_0$	C_n	$C_3 \vee C_2$
1	S - R	Same as R + S equations, but substitute \bar{R}_i for R_i in definitions			
2	R - S	Same as R + S equations, but substitute \bar{S}_i for S_i in definitions			
3	R \vee S	LOW	$P_2 P_1 P_0$	$\bar{P}_2 \bar{P}_1 \bar{P}_0 + C_n$	$\bar{P}_2 \bar{P}_1 \bar{P}_0 + C_n$
4	R \wedge S	LOW	$\bar{G}_3 + \bar{G}_2 + \bar{G}_1 + \bar{G}_0$	$G_3 + G_2 + G_1 + G_0 + C_n$	$G_3 + G_2 + G_1 + G_0 + C_n$
5	$\bar{R} \wedge \bar{S}$	LOW	Same as R \wedge S equations, but substitute \bar{R}_i for R_i in definitions		
6	R ∇ S	Same as R \vee S, but substitute \bar{R}_i for R_i in definitions			
7	$\bar{R} \nabla \bar{S}$	$G_3 + G_2 + G_1 + G_0$	$G_3 + P_2 \bar{G}_2 + P_2 P_2 \bar{G}_1 + P_2 P_2 P_2 \bar{G}_0$	$\bar{G}_3 + P_2 \bar{G}_2 + \bar{P}_2 \bar{P}_2 \bar{G}_1 + \bar{P}_2 \bar{P}_2 \bar{P}_2 \bar{G}_0 + \bar{P}_2 \bar{P}_2 \bar{P}_2 \bar{C}_n$	See note

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

HIGH transition. The clock LOW time is internally the write enable to the 16×4 RAM which comprises the "master" latches of the register stack. While the clock is LOW, the "slave" latches on the RAM outputs are closed, storing the data previously on the RAM outputs. This allows synchronous master-slave operation of the register stack.

Expansion of The Am2901

Any number of Am2901's can be interconnected to form CPU's of 12, 16, 24, 36, or more bits, in 4-bit increments. Figure 4 illustrates the interconnection of three Am2901's to form a 12-bit CPU, using ripple carry. Figure 5 illustrates a 16-bit CPU using carry lookahead, and Fig. 6 is the general carry lookahead scheme for long words.

With the exception of the carry interconnection, all expansion schemes are the same. The Q_3 and RAM_3 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 Q_0 and RAM_0 pins of the adjacent more significant device. These connections allow the Q registers of all Am2901'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 three-state multiplexers which can be controlled by the microcode to select the appropriate input signals to the shift inputs. (See Fig. 7.)

The open-collector $F = 0$ outputs of all the Am2901's are connected together and to a pull-up resistor. This line will go HIGH if and only if the output of the ALU contains all zeros. Most systems will use this line as the Z (zero) bit of the processor status word.

The overflow and F_8 pins are generally used only at the most significant end of the array, and are meaningful only when 2's complement signed arithmetic is used. The overflow pin is the Exclusive-OR of the carry-in and carry-out of the sign bit (MSB). It will go HIGH when the result of an arithmetic operation is a number requiring more bits than are available, causing the sign bit to be erroneous. This is the overflow (V) bit of the processor status word. The F_8 pin is the MSB of the ALU output. It is the sign of the result in 2's complement notation, and should be used as the negative (N) bit of the processor status word.

The carry-out from the most significant Am2901 (C_{n+4} pin) is the carry-out from the array, and is used as the carry (C) bit of the processor status word.

Carry interconnections between devices may use either ripple carry or carry lookahead. For ripple carry, the carry-out (C_{n+4}) of each device is connected to the carry-in (C_n) of the next more significant device. Carry lookahead uses the Am2901 lookahead carry generator. The scheme is identical with that used with the 74181/74182. Figures 5 and 6 illustrate single- and multiple-level lookahead.

Shift I/O Lines at the End of the Array

The Q-register and RAM left/right shift data transfers occur between devices over bidirectional lines. At the ends of the array, three-state multiplexers are used to select what the new inputs to the registers should be during shifting. Figure 7 shows two Am25LS253 dual four-input multiplexers connected to provide four shift modes. Instruction bit 17 (from the Am2901) is used to select whether the left-shift multiplexer or the right-shift multiplexer is active. (See Table 8.) The four shift modes in this example are:

- | | |
|--------|---|
| Zero | A LOW is shifted into the MSB of the RAM on a down shift. If the Q register is also shifted then a LOW is deposited in the Q-register MSB. If the RAM or both registers are shifted up LOWs are placed in the LSBs. |
| One | Same as zero but a HIGH level is deposited in the LSB or MSB. |
| Rotate | A single-precision rotate. The RAM MSB shifts into the LSB on a right shift and the LSB shifts into the MSB on a left shift. The Q register if shifted, will rotate in the same manner. |

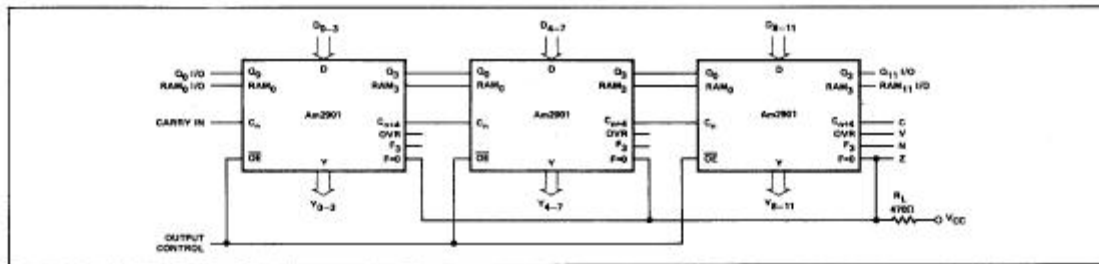


Fig. 4. Three Am2901's used to construct 12-bit CPU with ripple carry. Corresponding A, B, and 1 pins on all devices are connected together.

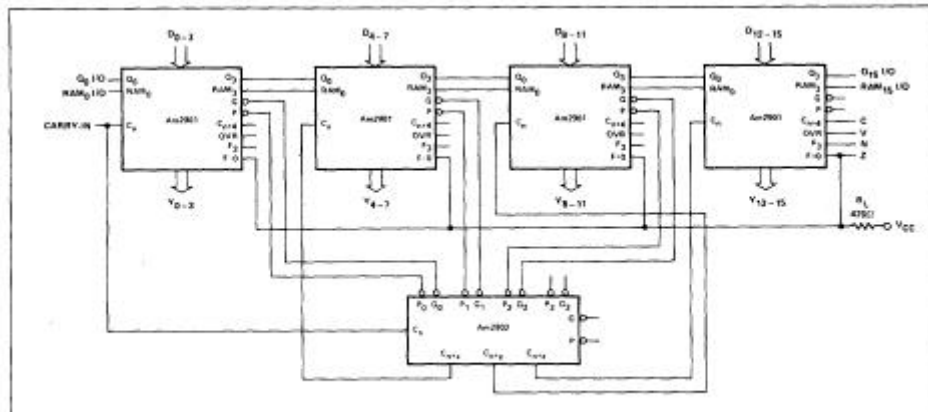


Fig. 5. Four Am2901's in a 16-bit CPU using the Am2902 for carry lookahead.

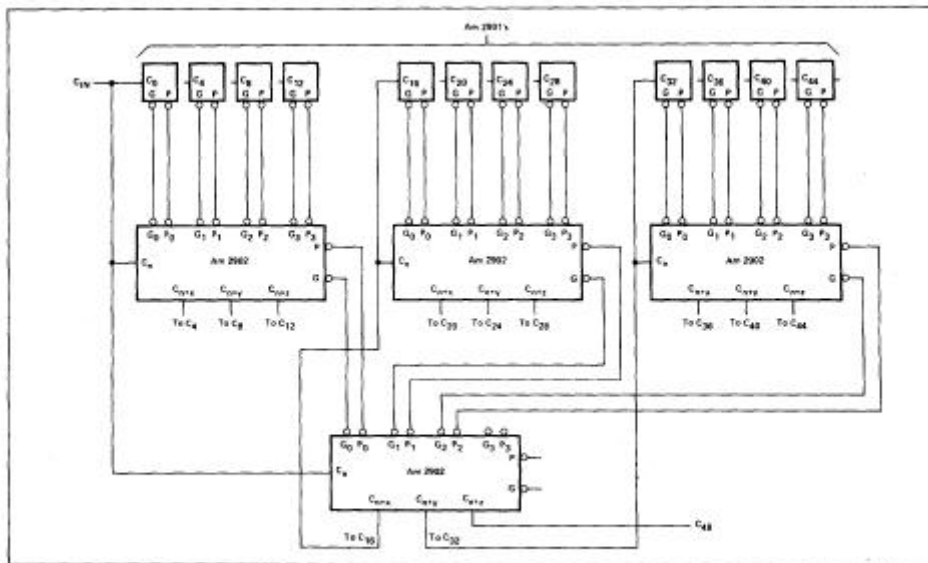


Fig. 6. Carry lookahead scheme for 48-bit CPU using 12 Am2901's. The carry-out flag (C48) should be taken from the lower Am2902 rather than the right-most Am2901 for higher speed.

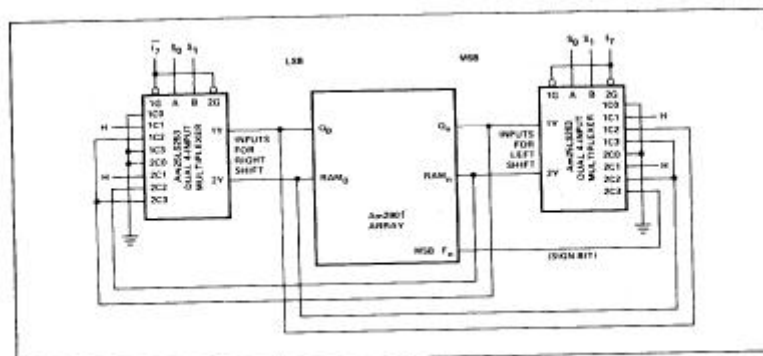


Fig. 7. Three-state multiplexers-used on shift I/O lines.

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)

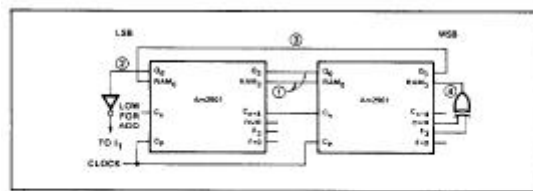
Figure 8 shows the connections for multiplication. The circled numbers refer to the paragraphs below.

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					Type
I_7	S_1	S_0	Q_0	Q_n	RAM_0	RAM_n	Shift
H	L	L	0	Q_{n-1}	0	F_{n-1}	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 Arithmetic
							Up (Right)

H	H	H	0	Q_{n-1}	Q_n	F_{n-1}	
L	L	L	Q_1	0	F_1	0	Zero
L	L	H	Q_1	1	F_1	1	One
L	H	L	Q_1	Q_0	F_1	F_0	Down (Left) Rotate Arithmetic
L	H	H	Q_1	F_0	F_1	$RAM_n = RAM_{n-1} = F_n$	



8-bit (or more) ALU output can be shifted as a unit prior to storage in the register stack.

3 As the new partial product appears at the input to the register stack, it is shifted left by the RAM shifter. The new LSB of the partial product, which is complete and will not be affected by future operations, is available on the RAM₀ pin. This signal is returned to the MSB of the Q register. On each cycle then, the just-completed LSB of the product is deposited in the MSB of the Q register; the Q register fills with the least significant half of the product.

$$Y = -Y_1 2^i + Y_{i-1} 2^{i-1} + \dots + Y_0 2^0$$

This scheme will produce a correct 2's complement product for all multiplicands and multipliers in 2's complement notation.

Figure 9 is a table showing the input states of the Am2901's for each step of a signed 2's complement multiplication.

Am2909 Microprogram Sequencer

General Description

The Am2909 is a 4-bit-wide address controller intended for sequencing through a series of microinstructions contained in a ROM or PROM. Two Am2909's may be interconnected to generate an 8-bit address (256 words), and three may be used to generate a 12-bit address (4096 words). Figure 10 is a block diagram of the Am2909.

The Am2909 can select an address from any of four sources. They are: (1) a set of external direct inputs (D); (2) external data from the R inputs, stored in an internal register; (3) a 4-

Initial Register States				Am2901 Microcode												Final Register States			
R				Program: 2's Comp. Multiply Date: 8/5/75 By: J.S.												R			
0	Multiplier															0	Multiplier		
1	Multiplicand															1	Multiplicand		
2	X															2	LSH Product		
3	X															3	MSH Product		
S, F	D	Description	Repeat	Pin States (Octal)												Jump			
Q V A	Q	Move Multiplier to Q	—	0	X	0	3	4	X	X	X	X	X	X	X				
Q V B	B	Clear R ₃	—	X	3	2	4	3	X	X	X	X	X	X	X				
(Q+B)/2 (A+B)/2	B	Cond. Add & Shift	n-1	1	3	4	0	1 or 3 I ₃ = Q ₃ LO	0	—	RAM ₀	—	F ₃ VOVR						
(B-Q)/2 (B-A)/2	B	Cond. Subt. & Shift	—	1	3	4	1	1 or 3 I ₁ = Q ₀ LO	1	—	RAM ₀	—	F ₃ VOVR						
Q V Q	B	Move LSH Prod. to R ₃	—	X	2	2	3	2	X	X	X	X	X	X	X				

Fig. 9

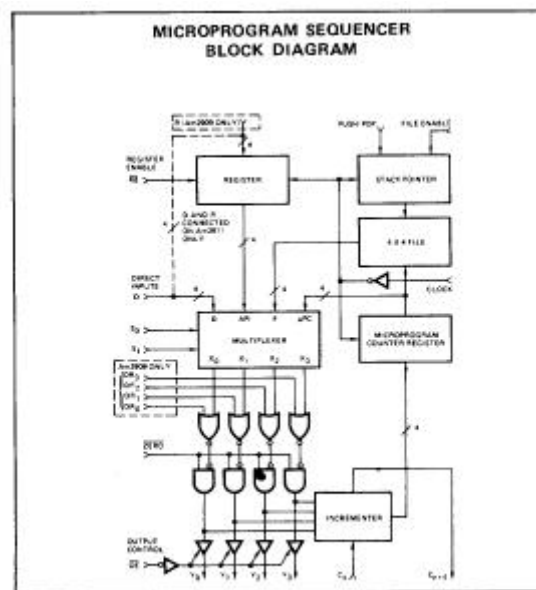
push/pop stack; or (4) a program counter register (which usually contains the last address plus one). The push/pop stack includes certain control lines so that it can efficiently execute nested subroutine linkages. Each of the four outputs can be ORed with an external input for conditional skip or branch 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 S₀ and S₁ 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 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.

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 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 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 micro instructions 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

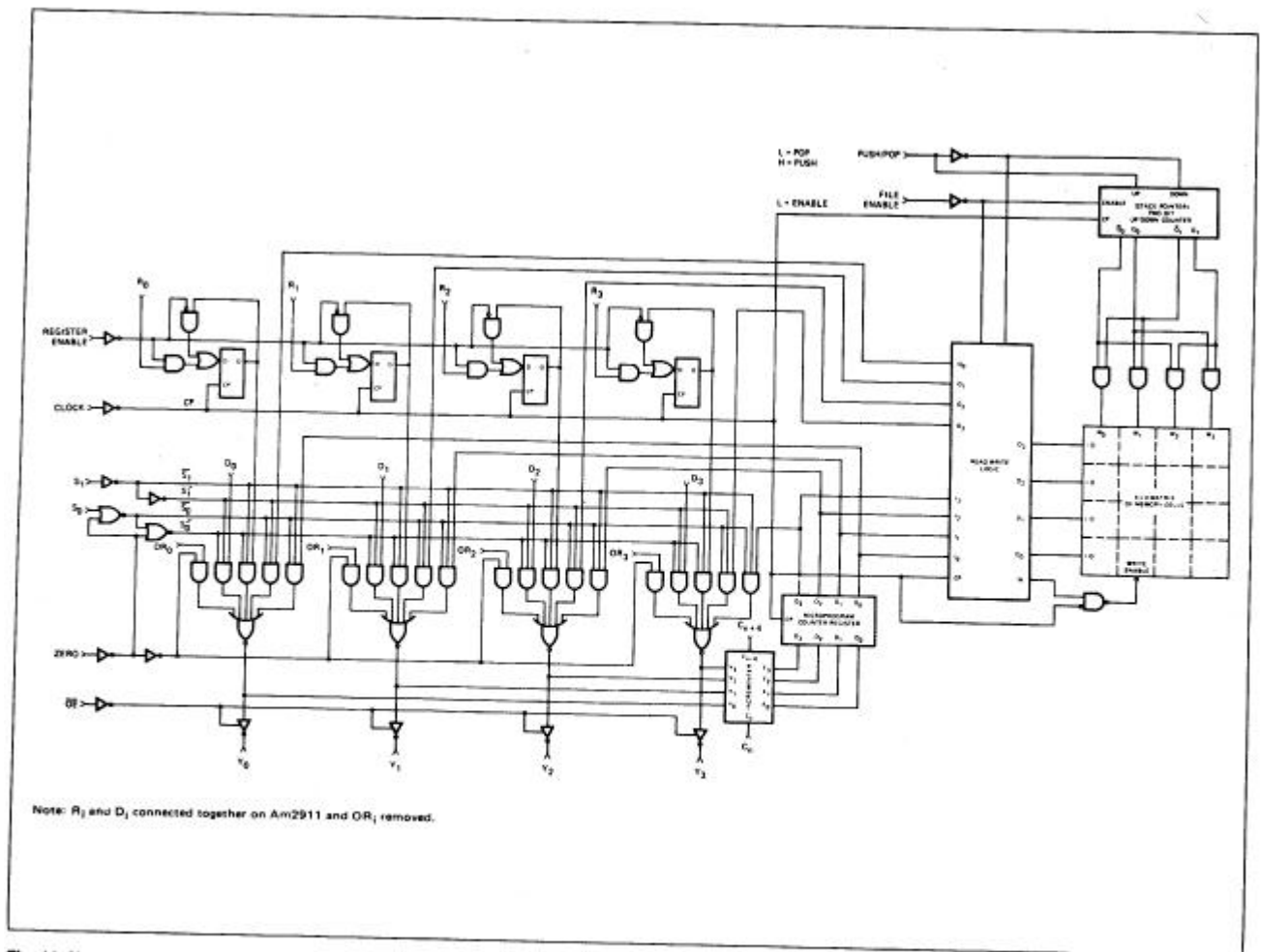


Fig. 11. Microprogram sequencer block diagram.

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

\underline{S}_1, S_0	Control lines for address source selection.
\underline{FE}, PUP	Control lines for push/pop stack.
RE	Enable line for internal address register.
\underline{OR}_i	Logic OR inputs on each address output line.
ZERO	Logic AND input on the output lines.
OE	Output enable. When 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 μ PC 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

μ PC	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.
μ WR	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 combinational logic for branching) determine which data source contains the

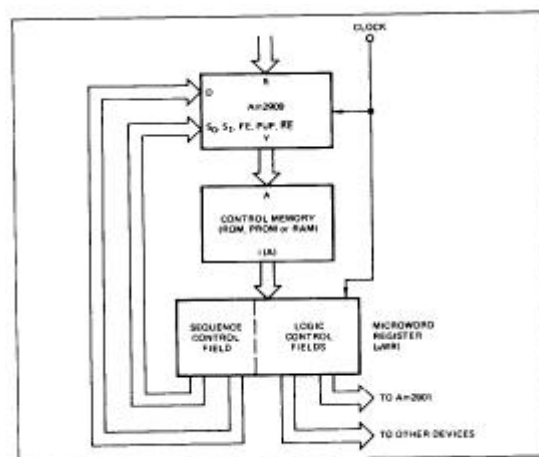


Fig. 12. Microprogram sequencer control.

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 , 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 , 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 .

Address Selection					Output Control				
OCTAL	S_1	S_0	SOURCE FOR Y OUTPUTS	SYMBOL	OR_i	ZERO	\overline{OE}	Y_i	
0	L	L	Microprogram Counter	μPC	X	X	H	Z	
1	L	H	Register	REG	X	L	L	L	
2	H	L	Push-Pop stack	STK0	H	H	L	H	
3	H	H	Direct inputs	D_i	L	H	L	Source selected by $S_0 S_1$	

Z = High Impedance

Synchronous Stack Control		
FE	PUP	PUSH-POP STACK CHANGE
H	X	No change
L	H	Increment stack pointer, then push current PC onto STK0
L	L	Pop stack (decrement stack pointer)

H = High

L = Low

X = Don't Care

Fig. 13

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

<i>Cycle</i>	<i>S_I, S_O, FE, PUP</i>	<i>μPC</i>	<i>REG</i>	<i>STK0</i>	<i>STK1</i>	<i>STK2</i>	<i>STK3</i>	<i>Y_{OUT}</i>	<i>Comment</i>	<i>Principal use</i>
N	0 0 0 0	J	K	Ra	Rb	Rc	Rd	J	Pop stack	End loop
N+1	J+1	K	Rb	Rc	Rd	Ra	...		
N	0 0 0 1	J	K	Ra	Rb	Rc	Rd	J	Push μ PC	Setup loop
N + 1	J + 1	K	J	Ra	Rb	Rc	...		
N	0 0 1 X	J	K	Ra	Rb	Rc	Rd	J	Continue	Continue
N+1	J+1	K	Ra	Rb	Rc	Rd	...		
N	0 1 0 0	J	K	Ra	Rb	Rc	Rd	K	Pop stack;	End loop
N + 1	K+ 1	K	Rb	Rc	Rd	Ra	...	Use AR for address	
N	0 1 0 1	J	K	Ra	Rb	Rc	Rd	K	Push μ PC;	JSR AR
N + 1	K+ 1	K	J	Ra	Rb	Rc	...	Jump to address in AR	
N	0 1 1 X	J	K	Ra	Rb	Rc	Rd	K	Jump to address in AR	JMP AR
N+ 1	K + 1	K	Ra	Rb	Rc	Rd	...		
N	1 0 0 0	J	K	Ra	Rb	Rc	Rd	Ra	Jump to address in STK0;	RTS
N+1	Ra+1	K	Rb	Rc	Rd	Rc	...	Pop stack	

N	1 0 0 1	J	K	Ra	Rb	Rc	Rd	R	Jump to address in STK0;	
N+1	Ra+1	K	J	Ra	Rb	Ra	...	Push μ PC	
N	1 0 1 X	J	K	Ra	Rb	Rc	Rd	Ra	Jump to address in STK0	Stack ref (loop)
N + 1	Ra+ 1	K	Ra	Rb	Rc	Rd	...		
N	1 1 0 0	J	K	Ra	Rb	Rc	Rd	D	Pop stack;	End loop
N + 1	D+1	K	Rb	Rc	Rd	Ra	...	Jump to address on D	
N	1 1 0 1	J	K	Ra	Rb	Rc	Rd	D	Jump to address on D	JSR D
N + 1	D + 1	K	J	Ra	Rb	Rc	...	Push μ PC	
N	1 1 1 X	J	K	Ra	Rb	Rc	Rd	D	Jump to address on D	JMP D
N + 1	D+1	K	Ra	Rb	Rc	Rd	...		

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

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

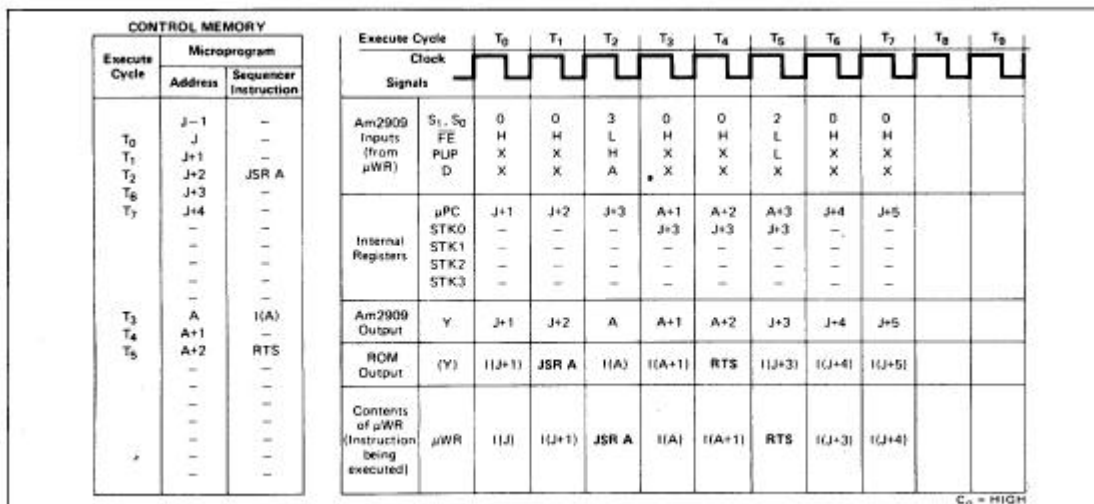


Fig. 14. Subroutine execution.

APPENDIX 1 AM2909 ISP DESCRIPTION



```

AMD901 :=
begin
! TSPS description of the AMD 901 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 refines the actual instruction execution process.
! Page 4 contains routines that aid in computation of the carry generate
! (G), overflow (OVR), and carry propagate (P).
! outputs.

**PC.State**
R(3:0) := 1 R inputs to ALU
F(3:0) := 1 S inputs to ALU
Q(3:0) := 1 Output from Q register

**MP.State**
RAM[8:15](3:0) := 16 8 x 4 bit 2 port RAM
A.LATCH(3:0) := 1 A RAM port latch
B.LATCH(3:0) := 1 B RAM port latch

**Internal.State**
ALC(3:0) := 1 A RAM port input address
RC(3:0) := 1 B RAM port input address
BC(3:0) := 1 Direct data inputs
IC(3:0) := 1 Data outputs
OE<?> := 1 Output enable (tristate control)
PC<?> := 1 Carry propagate
GC<?> := 1 Carry generate
OVR<?> := 1 Overflow
FEQLD<?> := 1 High if ALU output = 0
Ca<?> := 1 Carry in
Cc<?> := 1 Carry out
Ca4<?> := 1 Low order shift input/output
Cn4<?> := 1 High order shift input/output
DMG<?> := 1 Low order Q shift input/output
DO<?> := 1 High order Q shift input/output
QB<?> := 1 High order Q shift input/output

**Instruction.Format**
I(8:0) := 1 Instruction fields
Src(2:0) := I(2:0) Source operand field
Op(2:0) := I(5:3) Operation field
Dest(2:0) := I(6:0) Destination operand field

**Implementation Variables**
ALUC(3:0) := 1 ALU + carry output
cltmp(3:0) := 1 Temporary for generating carry
macro z := !'1111'.
Tristate constant

**Instruction.Cycle**(us)
start(name) := 1 Initialization
begin
OP <- OVR = FEQLD = Cn4 = 0;
P <- G = 1 next

run :=
begin
! Main instruction cycle
source(0) next
srcnc(1) next
dest(2:0)(3:0) next
RESTART run
end

and,

**Access.Computations**(us)
source := 1 Source calculation
begin
A.LATCH = RAM[A];
B.LATCH = RAM[B] next
RECODE src <-
begin
#0 := (R = A.LATCH); S = 0
#1 := (R = A.LATCH); S = B.LATCH
#2 := (R = 0); S = 0
#3 := (R = 0); S = B.LATCH
#4 := (R = 0); S = A.LATCH
#5 := (R = 0); S = A.LATCH
#6 := (R = D); S = 0
#7 := (R = D); S = 0
end
end

destination := 1 Destination calculation
begin
(OVER dest <-)
begin
#0 := ((R = F); S = F);
#1 := ((R = F);
#2 := (R = RAM[A]); RAM[S] = F);
#3 := (R = F); RAM[S] = F);
#4 := (R = F); RAM[S] = RAM[S] XOR RAM[D];
#5 := (R = F); RAM[S] = RAM[S] XOR RAM[D];
#6 := (R = F); RAM[S] = RAM[S] XOR RAM[D];
#7 := (R = F); RAM[S] = RAM[S] XOR RAM[D];
end

and,

**Instruction.Execution**(us)
exec :=
begin
DECODE op <-
begin
#0 := (R = R + S);
P = ((R or S) eq(us) '1111');
G = g.compute(R, S);
OVR = ALU(4) xor ALUC(3);
#1 := (ALU = S - R);
P = (((not R) or S) eq(us) '1111');
G = g.compute(not R, S);
OVR = ALU(4) xor ALUC(3);
#2 := (ALU = R - S);
P = ((R or (not S)) eq(us) '1111');
G = g.compute(R, (not S));
OVR = ALU(4) xor ALUC(3);
#3 := (ALU = R or S);
P = 1;
G = ((R or S) eq(us) '1111') next
Cn4 = OVR = (not G) or Ca;
#4 := (ALU = R and S);
P = 0;
G = ((R and S) eq(us) '1111') next
Cn4 = OVR = (not G) or Ca;
#5 := (ALU = ((not R) and S) eq(us) '1111') next
Cn4 = OVR = (not G) or Ca;
#6 := (ALU = ((not R) and S) eq(us) '1111') next
Cn4 = OVR = (not G) or Ca;
#7 := (ALU = ((not R) and S) eq(us) '1111') next
Cn4 = OVR = (not G) or Ca;
end

and,

**Service.Facilities**(us)
g.compute(r,(3:0), s,(3:0)) <-
begin
g.compute = (((r and s) eq(us) '1111') and ((r and s,(3:0)) eq(us) '000'))
or (((not r) and s) eq(us) '1111') and ((r and s,(3:0)) eq(us) '000'))
or (((r and not s) eq(us) '1111') and ((r and s,(3:0)) eq(us) '000'))
or (((not r and not s) eq(us) '1111') and ((r and s,(3:0)) eq(us) '000'))
end

and,

overf((r,(3:0), s,(3:0))) <-
begin
overf = ((r and s) eq(us) '1111') and ((r and s,(3:0)) eq(us) '000')
or (((not r) and s) eq(us) '1111') and ((r and s,(3:0)) eq(us) '000'))
or (((r and not s) eq(us) '1111') and ((r and s,(3:0)) eq(us) '000'))
or (((not r and not s) eq(us) '1111') and ((r and s,(3:0)) eq(us) '000'))
end

end
! End of AMD901 description
end

```

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 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, 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 E_A input selects either the DA external data input or RAM output port A for use as one ALU operand, and the 0E_B and I₀ 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 E_A , 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, G, and carry propagate, 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 ALL 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 G/N and P/OVR outputs indicate G and 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} , P/OVR, and G/N signals vary with the ALL 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.

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>I₄</i>	<i>I₃</i>	<i>I₂</i>	<i>I₁</i>	<i>Hex code</i>	<i>ALU functions</i>
<i>L</i>	<i>L</i>	<i>L</i>	<i>L</i>	<i>0</i>	$I_0 = H$ $F_i = \text{HIGH}$ $I_0 = L$ Special functions
<i>L</i>	<i>L</i>	<i>L</i>	<i>H</i>	<i>1</i>	$F = S \text{ Minus } R \text{ Minus } 1 \text{ Plus } C_n$
<i>L</i>	<i>L</i>	<i>H</i>	<i>L</i>	<i>2</i>	$F = R \text{ Minus } S \text{ Minus } 1 \text{ Plus } C_n$
<i>L</i>	<i>L</i>	<i>H</i>	<i>H</i>	<i>3</i>	$F = R \text{ Plus } S \text{ Plus } C_n$
<i>L</i>	<i>H</i>	<i>L</i>	<i>L</i>	<i>4</i>	$F = S \text{ Plus } C_n$
<i>L</i>	<i>H</i>	<i>L</i>	<i>H</i>	<i>5</i>	$F = \sim S \text{ Plus } C_n$
<i>L</i>	<i>H</i>	<i>H</i>	<i>L</i>	<i>6</i>	$F = R \text{ Plus } C_n$
<i>L</i>	<i>H</i>	<i>H</i>	<i>H</i>	<i>7</i>	$F = \sim R \text{ Plus } C_n$
<i>H</i>	<i>L</i>	<i>L</i>	<i>L</i>	<i>8</i>	$F_i = \text{LOW}$
<i>H</i>	<i>L</i>	<i>L</i>	<i>H</i>	<i>9</i>	$F_i = R_i \text{ AND } S_i$
<i>H</i>	<i>L</i>	<i>H</i>	<i>L</i>	<i>A</i>	$F_i = R_i \text{ Exclusive-NOR } S_i$
<i>H</i>	<i>L</i>	<i>H</i>	<i>H</i>	<i>B</i>	$F_i = R_i \text{ Exclusive-OR } S_i$
<i>H</i>	<i>H</i>	<i>L</i>	<i>L</i>	<i>C</i>	$F_i = R_i \text{ AND } S_i$
<i>H</i>	<i>H</i>	<i>L</i>	<i>H</i>	<i>D</i>	$F_i = R_i \text{ NOR } S_i$
<i>H</i>	<i>H</i>	<i>H</i>	<i>L</i>	<i>E</i>	$F_i = R_i \text{ NAND } S_i$
<i>H</i>	<i>H</i>	<i>H</i>	<i>H</i>	<i>F</i>	$F_i = R_i \text{ OR } S_i$

L=LOW H=HIGH i= 0 to 3

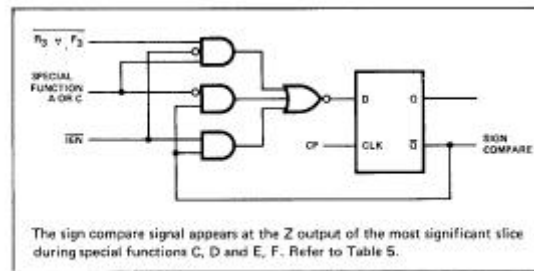


Fig. 2

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 $\overline{\text{OE}}_Y$ input is LOW and are in the high-impedance state when $\overline{\text{OE}}_Y$ is HIGH. Likewise, the DB output buffers are enabled when the $\overline{\text{OE}}_B$ is LOW and in the high-impedance state when $\overline{\text{OE}}_B$ 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 Y_{0-3} pins are all LOW, whether they are driven from the Y output buffers or from an external source connected to the Y_{0-3} 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, I_{0-8} ; the Instruction Enable input, IEN the LSS input; and the WRITE/ MSS input/output.

The 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 WRITE output as a function of the Am2903 instruction inputs.

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

When IEN is LOW, the 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 LSS input LOW programs the slice to operate as a least significant slice (LSS) and enables the WRITE output signal onto the WRITE/MSS bidirectional I/O pin. When LSS is tied HIGH, the WRITE/MSS pin becomes an input pin. Tying the WRITE/ 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

A₀₋₃ Four RAM address inputs which contain the address of the RAM word appearing at the RAM A output port.

B₀₋₃ Four RAM address inputs which contain the address of the RAM word appearing at the RAM B

Table 3 ALU Destination Control for I₀ OR I₁ OR I₂ OR I₃ OR I₄ HIGH, IEN = LOW

						SIO ₃		Y ₃	
<i>I₈</i>	<i>I₇</i>	<i>I₆</i>	<i>I₅</i>	<i>Hex code</i>	<i>ALU shifter function</i>	<i>Most sig. slice</i>	<i>Other slices</i>	<i>Most sig. slice</i>	<i>Other slices</i>
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

$$\text{Parity} = F_3 \vee F_2 \vee F_1 \vee F_0 \vee SIO_3$$

\vee = Exclusive OR

output port and into which new data is written when the WE input and the CP input are LOW

WE	The RAM write enable input. If WE is LOW, data at the Y I/O port is written into the RAM when the CP input is LOW. When 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.
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 OEB 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.

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.
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 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.
G /N	A multi-purpose pin which indicates the carry generate, 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.
P /OVR	A multi-purpose pin which indicates the carry

	Y ₂	Y ₁	Y ₀	SIO ₀	Write	Q Reg. & shifter function	QIO ₃	QIO ₀
<i>Most sig. slice</i>	<i>Other slices</i>							
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 → Q	Input	Q ₀
F ₃	F ₃	F ₂	F ₁	F ₀	L	Log. Q/2 → 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 → Q	Input	Q ₀
F ₂	F ₂	F ₁	F ₀	Parity	H	F → Q	Hi-Z	Hi-Z
F ₂	F ₂	F ₁	F ₀	Parity	L	F → 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 → Q	Q ₃	Input

F ₁	F ₁	F ₀	SIO ₀	Input	L	Log. 2Q → 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 → 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

H = HIGH

Hi-Z = high-impedance

propagate, 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 Y₀₋₃ 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.

SIO₀, SIO₃ Bidirectional serial shift inputs/outputs for the ALU shifter. During a shift-up operation, SIO₀ is an input and SIO₃ an output. During a shift-down operation, SIO₃ is an input and SIO₀ is an output. Refer to Tables 3 and 4 for an exact definition of these pins.

QIO₀, QIO₃ Bidirectional serial shift inputs/outputs for the Q shifter which operate like SIO₀ and SIO₃. Refer to Tables 3 and 4 for an exact definition of these pins.

LSS 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 WRITE output onto the WRITE/MSS pin. When LSS is tied HIGH, the chip is programmed to operate as either an intermediate or most significant slice and the WRITE output buffer is disabled.

WRITE/MSS When LSS is tied LOW, the WRITE output signal appears at this pin; the WRITE signal is LOW when an instruction which writes data into the RAM is being executed. When LSS is tied HIGH, 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).

Y₀₋₃ Four data inputs/outputs of the Am2903. Under control of the 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_3	I_7	I_6	I_5	Hex code	Special function	ALU function	ALU shifter function	SIO_3		SIO_0	Q Reg. & shifter function	QIO_3	QIO_2	WRITE
								Most sig. slice	Other slices					
L	L	L	X	0, 1	Unsigned Multiply	$F = S + C_0$ if $Z = L$ $F = R + S + C_0$ if $Z = H$	Log. $F/2 \rightarrow Y$ (Note 1)	Hi-Z	Input	F_2	Log. $Q/2 \rightarrow Q$	Input	Q_3	L
L	L	H	X	2, 3	Two's Complement Multiply	$F = S + C_0$ if $Z = L$ $F = R + S + C_0$ if $Z = H$	Log. $F/2 \rightarrow Y$ (Note 2)	Hi-Z	Input	F_2	Log. $Q/2 \rightarrow Q$	Input	Q_0	L
L	H	L	L	4	Increment by One or Two	$F = S + 1 + C_0$	$F \rightarrow Y$	Input	Input	Parity	Hold	Hi-Z	Hi-Z	L
L	H	L	H	5	Sign/Magnitude- Two's Complement	$F = S + C_0$ if $Z = L$ $F = S + C_0$ 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	$F = S + C_0$ if $Z = L$ $F = S - R - 1 + C_0$ if $Z = H$	Log. $F/2 \rightarrow Y$ (Note 2)	Hi-Z	Input	F_2	Log. $Q/2 \rightarrow Q$	Input	Q_0	L
H	L	L	X	8, 9	Single Length Normalize	$F = S + C_0$	$F \rightarrow Y$	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.	$F = S + C_0$	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	$F = S + R + C_0$ if $Z = L$ $F = S - R - 1 + C_0$ if $Z = H$	Log. $2F \rightarrow Y$	$R_0 \nabla F_3$	F_3	Input	Log. $2Q \rightarrow Q$	Q_0	Input	L
H	H	H	X	E, F	Two's Complement Divide, Correction and Remainder	$F = S + R + C_0$ if $Z = L$ $F = S - R - 1 + C_0$ 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 $C_{0,1}$ signal is internally gated to the Y_3 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.

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

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

Table 5

(Hex) $I_8I_7I_6I_5$		(Hex) $I_4I_3I_2I_1$	I_0	GI ($I = 0$ to 3)	PI ($I = 0$ to 3)	C_{n+4}	FOUR		GIN			Z	
							Most sig. slice	Other slices	Most sig. slice	Other slices	Most sig. slice	Intermediate slice	Least sig. slice
X	0	H	0	1	1	0	0	0	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	1	X	$\bar{R}_1 \wedge S_1$	$\bar{R}_1 \vee S_1$	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	2	X	$R_1 \wedge S_1$	$R_1 \vee S_1$	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	3	X	$R_1 \wedge S_1$	$R_1 \vee S_1$	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	4	X	0	S_1	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	5	X	0	S_1	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	6	X	0	R_1	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	7	X	0	\bar{R}_1	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	8	X	0	1	0	0	0	0	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	9	X	$\bar{R}_1 \wedge S_1$	1	0	0	0	0	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	A	X	$R_1 \wedge S_1$	$R_1 \vee S_1$	0	0	0	0	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	B	X	$\bar{R}_1 \wedge S_1$	$\bar{R}_1 \vee S_1$	0	0	0	0	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	C	X	$R_1 \wedge S_1$	1	0	0	0	0	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	D	X	$\bar{R}_1 \wedge S_1$	1	0	0	0	0	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	E	X	$R_1 \wedge S_1$	1	0	0	0	0	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
X	F	X	$\bar{R}_1 \wedge S_1$	1	0	0	0	0	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
0, 1	0	L	0 if $Z = L$ $R_1 \wedge S_1$ if $Z = H$	S_1 if $Z = L$ $R_1 \vee S_1$ if $Z = H$	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_3	\bar{G}	Input	Input	Input	Q_0
2, 3	0	L	0 if $Z = L$ $R_1 \wedge S_1$ if $Z = H$	S_1 if $Z = L$ $R_1 \vee S_1$ if $Z = H$	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_3	\bar{G}	Input	Input	Input	Q_0
4	0	L	See Note 1	See Note 2	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_3	\bar{G}	$\bar{Y}_8\bar{Y}_7\bar{Y}_6\bar{Y}_5$	$\bar{Y}_4\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$	$\bar{Y}_0\bar{Y}_3\bar{Y}_2\bar{Y}_1$
5	0	L	0	S_1 if $Z = L$ S_1 if $Z = H$	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_3 if $Z = L$ $F_3 \nabla S_1$ if $Z = H$	\bar{G}	S_1	Input	Input	Input
6, 7	0	L	0 if $Z = L$ $R_1 \wedge S_1$ if $Z = H$	S_1 if $Z = L$ $R_1 \wedge S_1$ if $Z = H$	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_3	\bar{G}	Input	Input	Input	Q_0
8, 9	0	L	0	S_1	See Note 3	$Q_2 \nabla Q_1$	\bar{P}	Q_1	\bar{G}	$\bar{Q}_0\bar{Q}_1\bar{Q}_2\bar{Q}_3$	$\bar{Q}_0\bar{Q}_1\bar{Q}_2\bar{Q}_3$	$\bar{Q}_0\bar{Q}_1\bar{Q}_2\bar{Q}_3$	$\bar{Q}_0\bar{Q}_1\bar{Q}_2\bar{Q}_3$
A, B	0	L	0	S_1	See Note 4	$F_2 \nabla F_1$	\bar{P}	F_2	\bar{G}	See Note 5	See Note 5	See Note 5	See Note 5
C, D	0	L	$R_1 \wedge S_1$ if $Z = L$ $R_1 \wedge S_1$ if $Z = H$	$R_1 \vee S_1$ if $Z = L$ $R_1 \wedge S_1$ if $Z = H$	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_2	\bar{G}	Sign Compare FF Output	Input	Input	Input
E, F	0	L	$R_1 \wedge S_1$ if $Z = L$ $R_1 \wedge S_1$ if $Z = H$	$R_1 \vee S_1$ if $Z = L$ $R_1 \vee S_1$ if $Z = H$	$G \vee PC_n$	$C_{n+3} \nabla C_{n+4}$	\bar{P}	F_2	\bar{G}	Sign Compare FF Output	Input	Input	Input

Notes: 1. If LSS is LOW, $G_0 = S_0$ and $G_{1,2,3} = 0$

Notes: 1. If LSS is LOW, $G_0 = S_0$ and $G_{12,11} = 0$
 If LSS is HIGH, $G_{12,11} = 0$
 2. If LSS is LOW, $P_0 = 1$ and $P_{12,11} = S_{12,11}$
 If LSS is HIGH, $P_0 = S_0$
 3. At the most significant slice, $C_{n+1} = Q_2 \nabla Q_1$
 At other slices, $C_{n+1} = G \vee PC_n$
 4. At the most significant slice, $C_{n+1} = F_2 \nabla F_1$
 At other slices, $C_{n+1} = G \vee PC_n$
 5. $Z = Q_0\bar{Q}_1\bar{Q}_2\bar{Q}_3$

L = LOW = 0
 H = HIGH = 1
 \wedge = OR
 \vee = AND
 ∇ = EXCLUSIVE OR
 $P = P_0P_1P_2P_3$
 $G = G_0 \vee G_1P_1 \vee G_2P_2 \vee G_3P_3$
 $C_{n+1} = G_0 \vee G_1P_1 \vee G_2P_2 \vee G_3P_3$

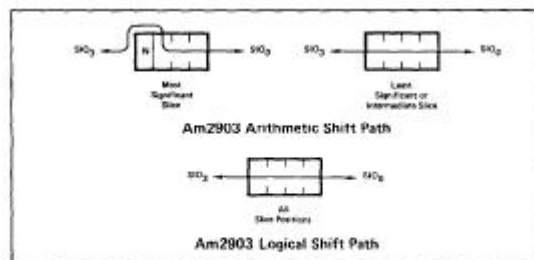


Fig. 3. Sign compare flip-flop.

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.

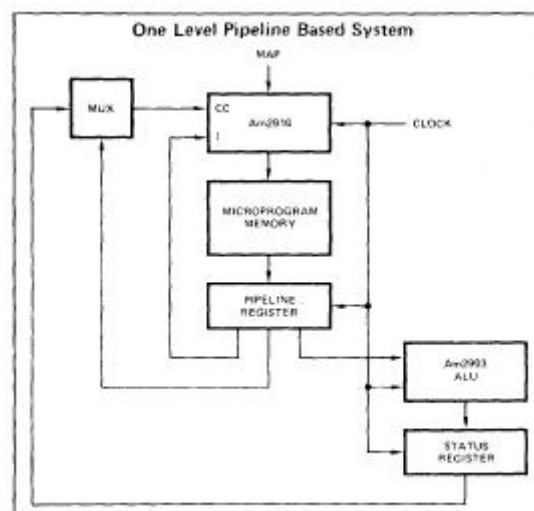


Fig. 4. Typical microprogram architecture.

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

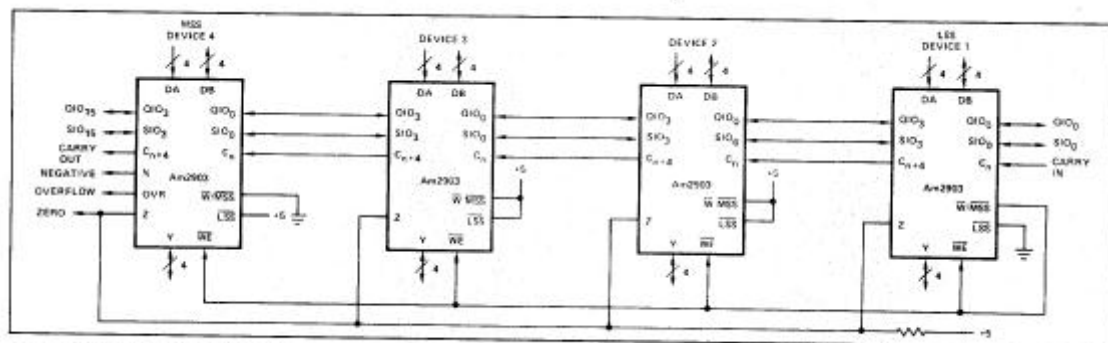


Fig. 5. 16-bit CPU with ripple carry.

most significant slice (MSS) with the LSS pin tied HIGH and the 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 G and P outputs of the Am2903 are connected to the appropriate G and 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 G/N and P/OVR pin functions are device-dependent. The most significant slice outputs N and OVR while all other slices output G and P.

The IEN pin of the Am2903 allows the option of conditional instruction execution. If IEN is LOW, all internal clocking is enabled, allowing the latches, RAM, and Q register to function, if IEN is HIGH, the RAM and Q register are disabled. The RAM is controlled by IEN if WE is connected to the 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 $A + B \rightarrow C$.

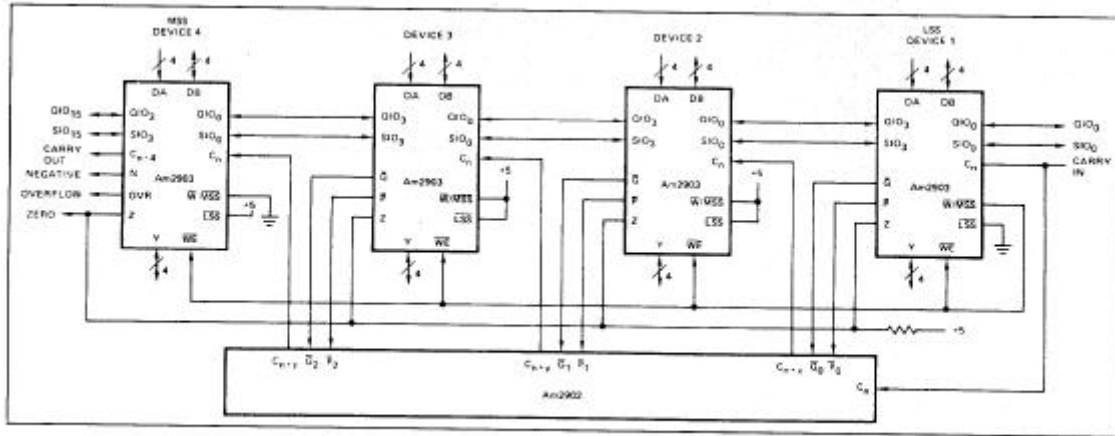


Fig. 6. 16-bit CPU with carry look ahead.

Implementation of a three-address architecture is made possible by varying the timing of LEN 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 $SIO_0 = F_{15} \vee F_{14} \vee F_{13} \vee \dots \vee F_1 \vee F_0 \vee SIO_{15}$.

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 QIO₀ 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} MSS = Q₃ MSS \vee Q₂ MSS).

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

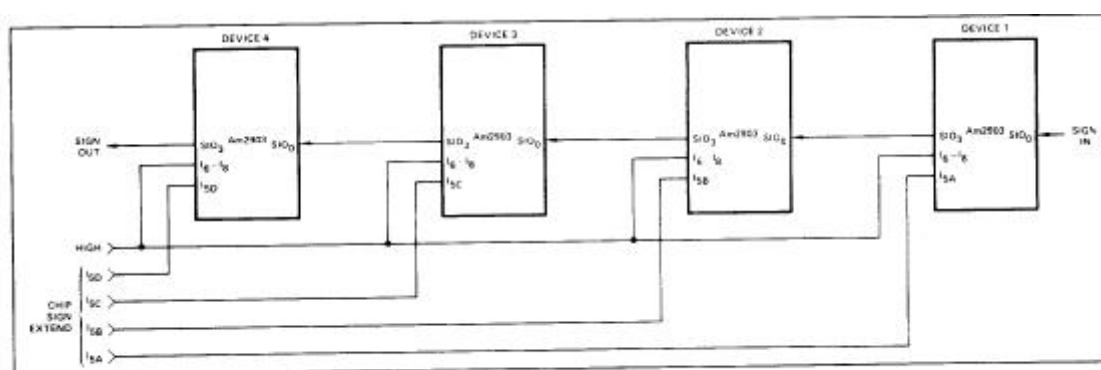


Fig. 7. Sign extend.

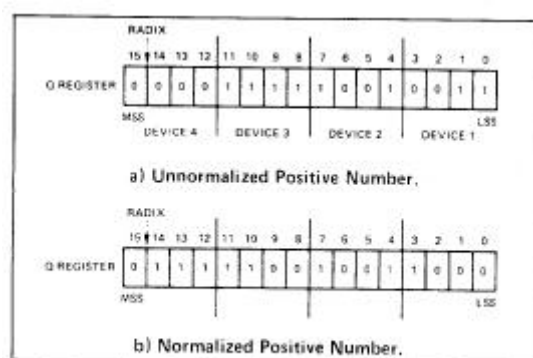


Fig. 8

available on the C_{n+4} pin (OVR = Q₂ MSS \vee Q₁ 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

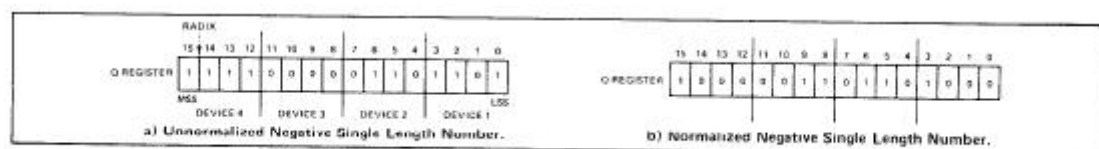


Fig. 9

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-

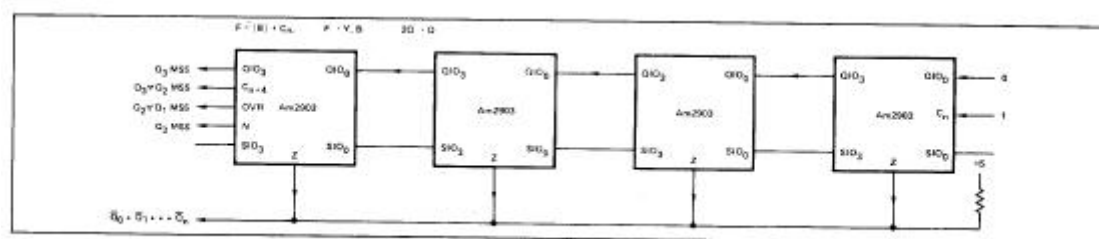


Fig. 10. Single-length normalize.

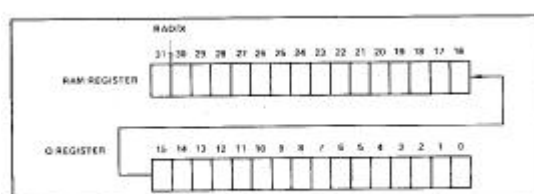


Fig. 11. Double-length word.

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 BAM 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

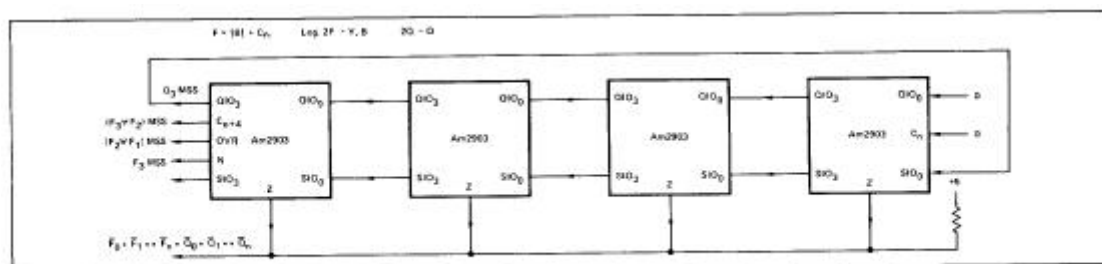


Fig. 12. Double-length normalize.

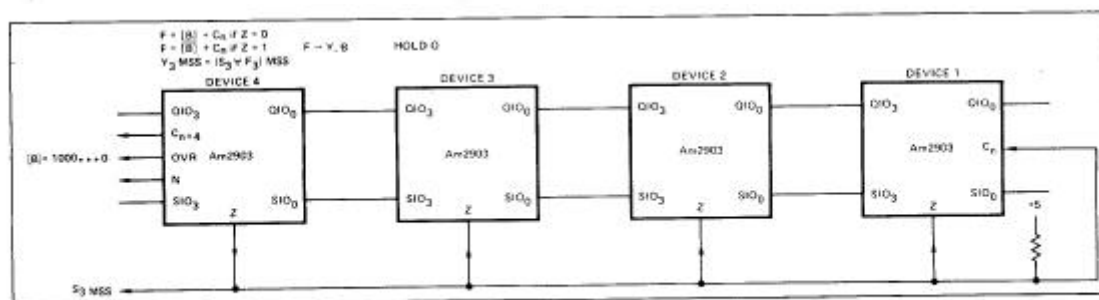


Fig. 13. 2's complement to sign/magnitude.

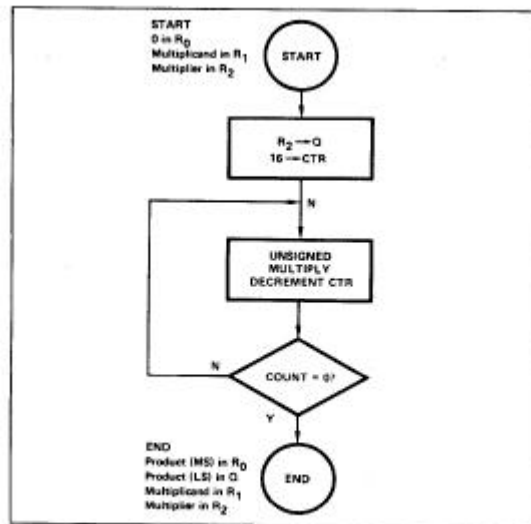


Fig. 14. 16x16 multiply flowchart.

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

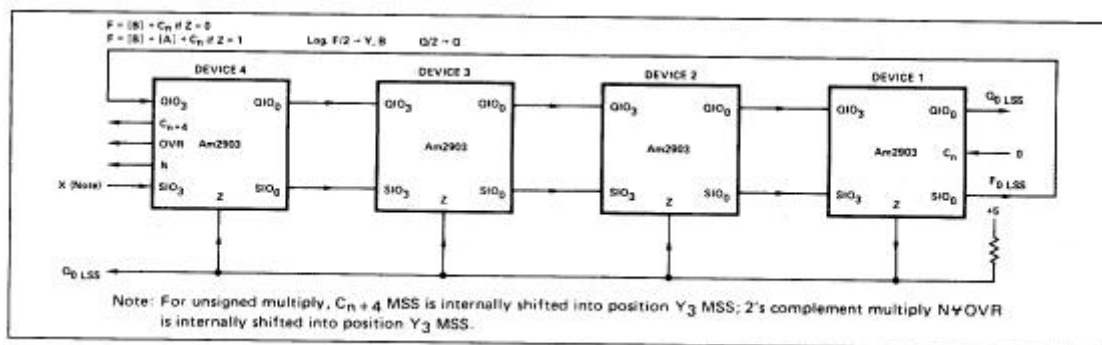


Fig. 15. Multiply.

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

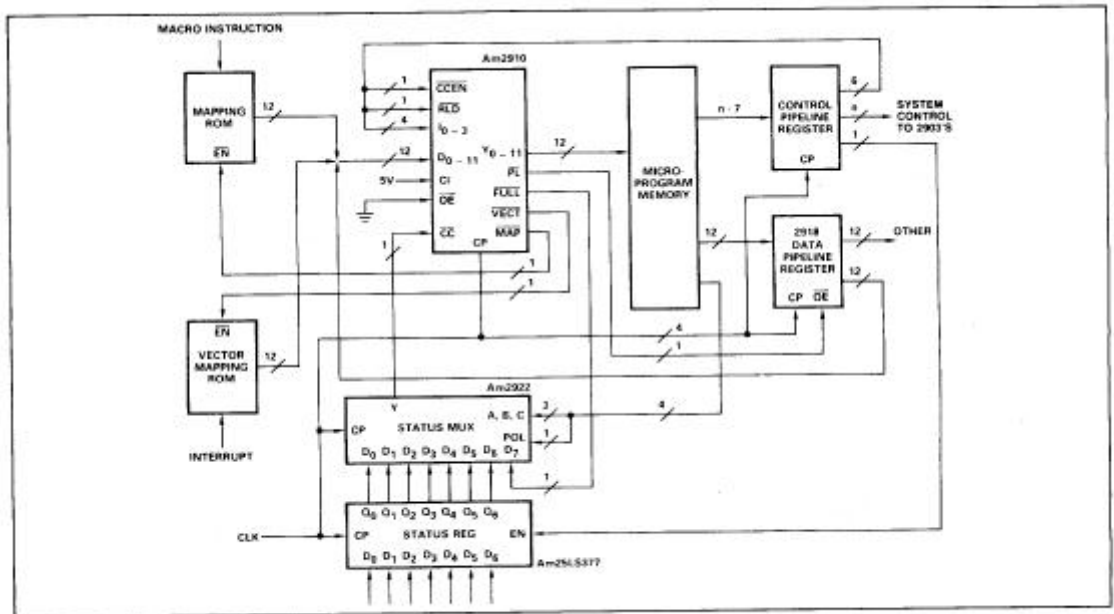


Fig. 16. Typical computer control unit (CCU).

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

Micro Memory Address	Inst	Data Pipeline Reg.	I ₀	I ₄ -I ₁	I ₈ -I ₅	OEB	OEV	A ₃ -A ₀	B ₃ -B ₀	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.

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-

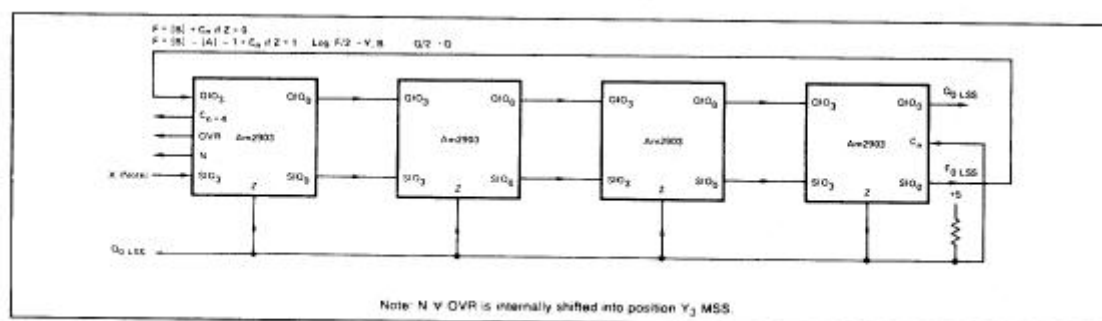


Fig. 18. 2's complement multiply, correction.

Memory Address	Am2910 Inst	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	00E10	X	6	6	X	X	R ₂	X	0									Load Counter & R ₂ → Q ₀
n+1	RPCT	n+1	0	0	2	0	0	R ₁	R ₀	0									2's Complement Multiply
n+2	X	X	0	0	6	0	0	R ₁	R ₀	Z									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.

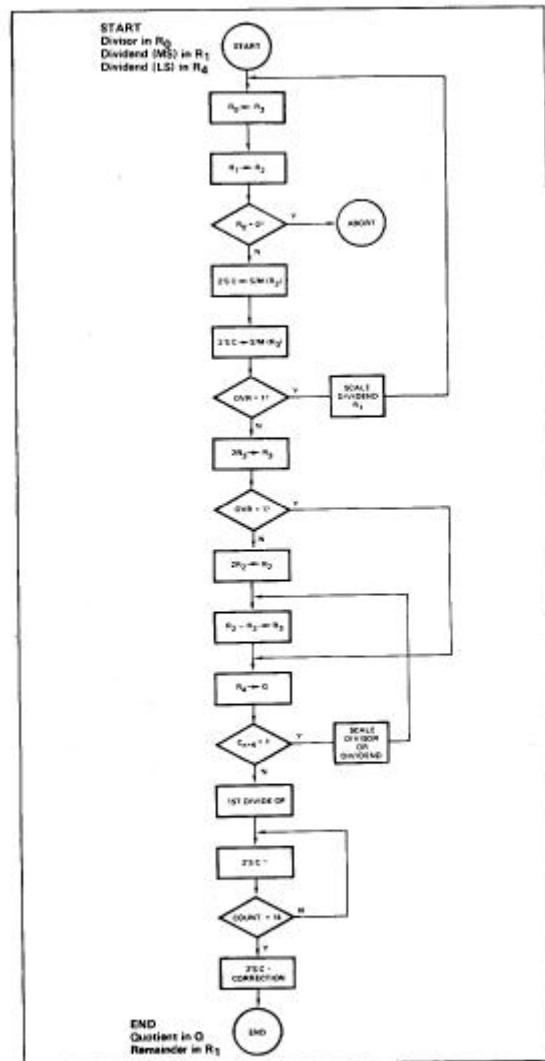


Fig. 20. Division flow chart—double precision divide.

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.

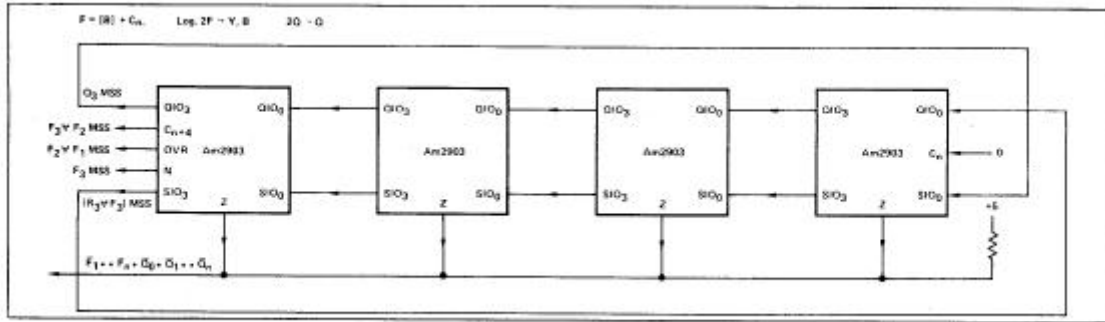


Fig. 21. Double-length normalize/first divide operation.

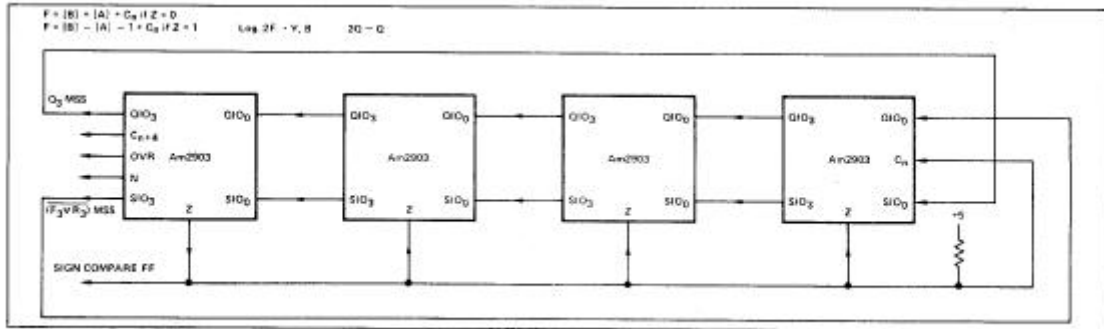


Fig. 22. 2's complement divide.

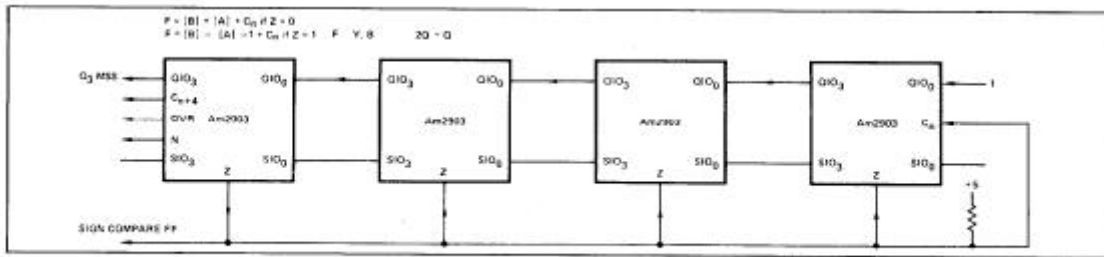


Fig. 23. 2's complement divide correction.

Am29LS18													
Micro Memory Address	Am2910 Inst.	Data Pipeline Reg.	Am2903							Am2922		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	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 $F = A$ 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. A_{0-3} address 16 memory locations in each chip while address bits A_{4-6} are decoded and used

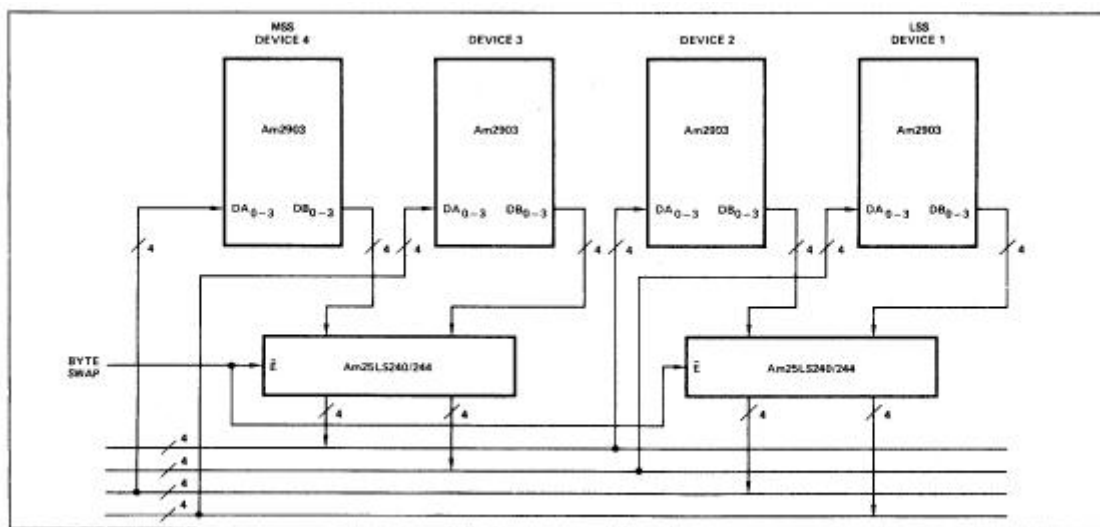


Fig. 25. Byte swap.

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 B_{0-3} are for source register addressing in each chip. Bits B_4 and B_5 are used for chip output enable selection. B_{6-9} access the 16 destination addresses on each chip, while bits B_{10}

and B_{11} 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 (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 B_4 and B_5 . 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 HAM address lines. When the destination address stabilizes on the address lines, the IEN pin is brought LOW. The WRITE output of the Am2903 will now go LOW, enabling the decoder sourced by address bits B_{10} and B_{11} . 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., $B_{0-3} = B_{6-9}$. 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 (μ PC), 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).

[illegible]

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 (μ PC) that is composed of a 12-bit incrementer followed by a 12-bit register. The μ PC 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 μ PC 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

28

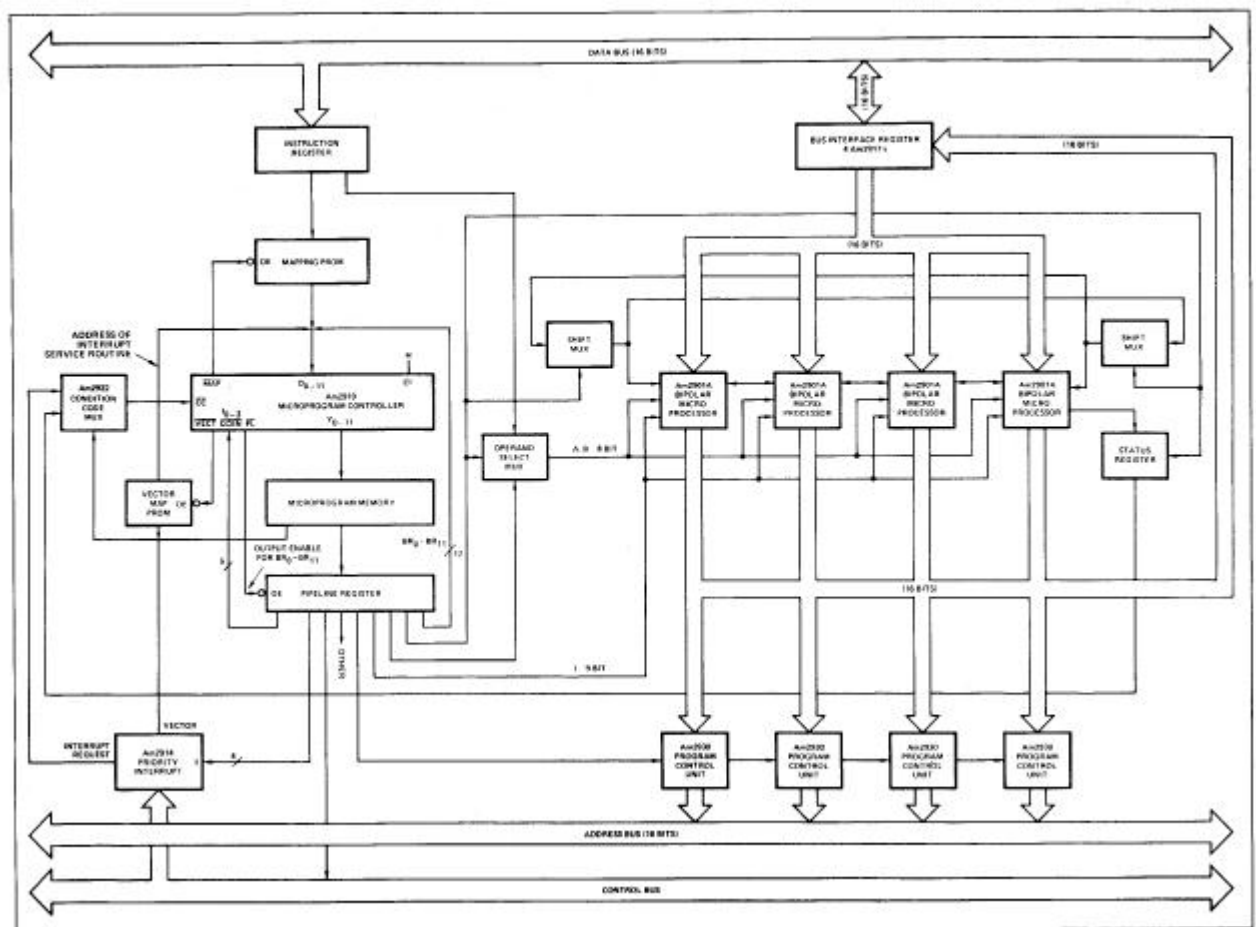


Fig. 28. Typical bipolar microcomputer using Am2910.

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 PL, MAP, and 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 μ PC 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 PL, MAP, and 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. 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 CC. If the 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 CC may be disabled for a specific microinstruction by setting CCEN HIGH, which unconditionally forces the action specified in the name; that is, it forces a pass. Other ways of using 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) unconditionally 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 ₅ -I ₀	Mnemonic	Name	Reg/ cntr contents	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

†† $\overline{CCEN} = \text{LOW}$ 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.
CC	Condition Code	Used as test criterion. Pass test is a LOW on CC.
CCEN	Condition Code Enable	Whenever the signal is HIGH, CC is ignored and the part operates as though CC were true (LOW).
CI	Carry-In	Low order carry input to incrementer for microprogram counter.
RLD	Register Load	When LOW forces loading of register/counter regardless of instruction or condition.
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.
FULL	FULL	Indicates that five items are on the stack.
PL	Pipeline Address Enable	Can select #1 source (usually Pipeline Register) as direct input source.
MAP	Map Address Enable	Can select #2 source (usually Mapping PROM or PLA) as direct input source.
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 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 (BR₀-BR₁₁ 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.

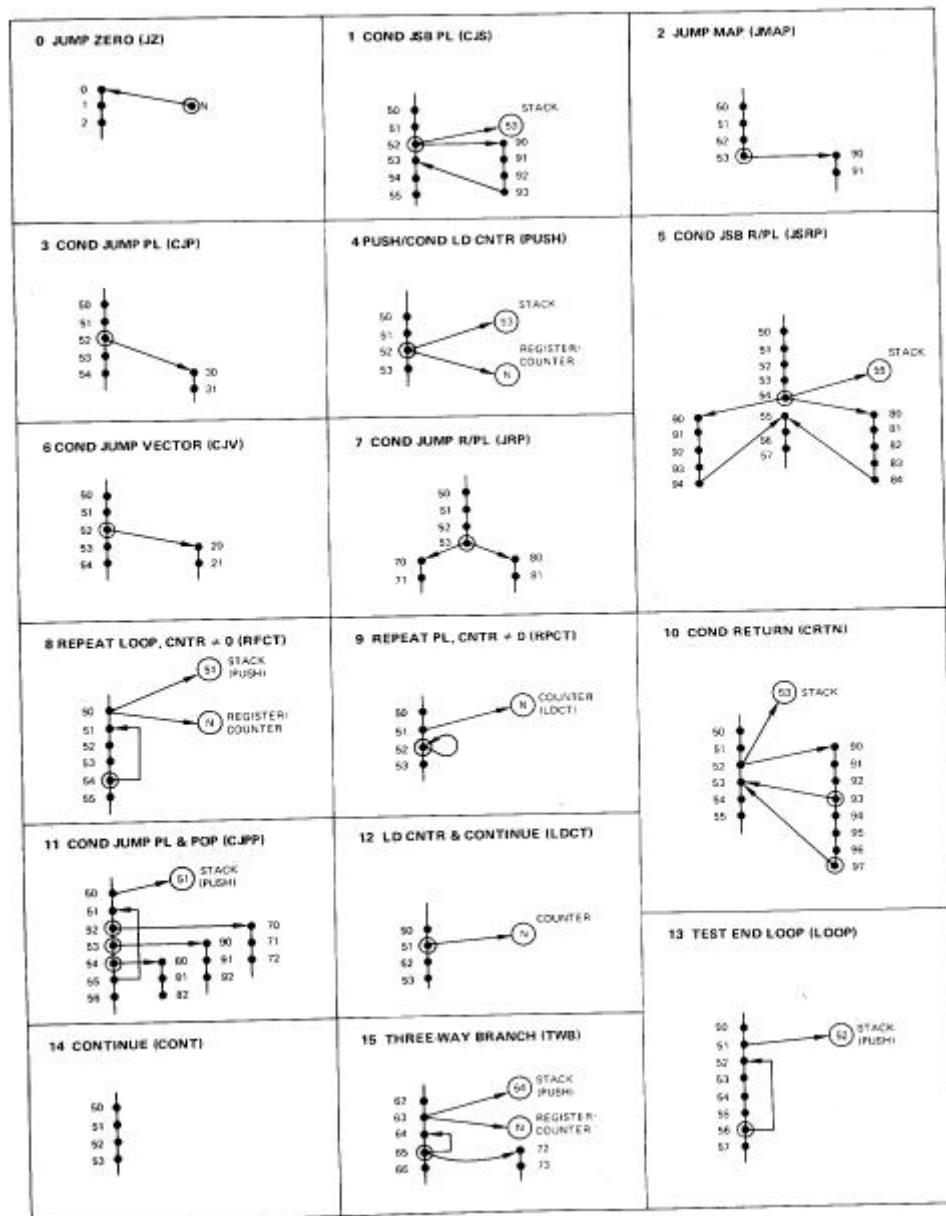


Fig. 29. Am2910 execution examples.

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 μ PC; 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 maybe 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 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 12, the conditional load included as part of instruction 4, and the use of the RLD input along with any instruction. The use of 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 12 is exactly equivalent to the combination of instruction 14 and 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 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 5, 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.


```

AM9083 :=
begin
! ISPS description of the AM9083 4 bit slice microprocessor.

! Page 1 of the description contains declarations of simple carriers
**PC.State** and **MP.State** sections describe the actual
carriers contained within the AM9083 chip.
**Internal.State** describes the simple carriers that
terminate in place.
**Implementation.Declarations** describe carriers necessary
for the ISP description.

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

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

! Page 5 - 9 contain descriptions of computations for the Z, G, A,
P.DNS, and Cnt Output pins plus the "q1" and "q2"
intermediate carry generate and carry propagate computations.

**PC.State**
ASHFT(CB),      ! ALU shifter
PC(B),          ! R inputs to ALU
S(B),           ! S inputs to ALU

**MP.State**
RAM[0:15](CB),   ! 16 X 4 bit 2 port RAM

**External.State**
A(CB),          ! A RAM port input address
B(CB),          ! B RAM port input address
CarryIn,        ! Carry In
DA(CB),         ! Direct data input (R input)
DB(CB),         ! Direct data input (S input)
SA(CB),         ! Sign of ALU (MSS)
F(CB),          ! Output from ALU
IEN(CB),        ! Instruction enable (IOM true)
LSSL(CB),       ! IOM == least Significant Slice
OEB(CB),        ! IOM enables RAM port B
OE(CB),         ! Output from Q register
Q100(CB),       ! Q register shift MSB
Q100(CB),       ! Q register shift LSB
S100(CB),       ! ALU shift MSB
W.MSS(CB),      ! ALU shift LSB
W.CB,           ! LSS+LOW => NOT WRITE output
Y(CB),          ! LSS+HIGH => input giv
Z(CB),          ! HIGH => LS LOW => MSS
IEN(CB),        ! Write Enable; LOW => RAM = Y
ICB(B),         ! Data Input/output
ICB(B),         ! Instruction Inputs

**Implementation.Declarations**
pc(),            ! Accumulator for computed P
g(),             ! Accumulator for computed G
cn(),            ! Accumulator for computed Cn's
writef(),        ! Internal write flag

dest(CB) := ICB(B),    ! Destination select
op(CB) := IC(L),       ! Function OP code
src() := IC(B),        ! IC's (part of source select)

macro bix := [1111],   ! Tri-state constant
macro parity := {FCB xor FCB xor FC(B) xor FC(B) xor S103},
macro mss := [LSL and (not W.MSS)],

**Access.Computation**[us]
source :=
begin
DECODE EA & S103 & OEB =>
begin
WD := (R + RAM[A]; S = RAM[B]),
RT := (T + ram[a]; S = B),
#2:#3 := (S + RAM[A]; S = Q),
#4 := (S + OA ; S = RAM[B]),
#5 := (R + OA ; S = OB ),
#6:#7 := (R + OA ; S = Q )
end
end

destination :=      ! Destination calculation
begin
IN(GBR CB);? =>
begin
Q := begin
! ASHFT= F/2

```

! Special functions are decoded from ICB(6) when ICB(0) equal zero.
! These are the built-in multiplication, division, and normalization
! functions.

```
special_functions :=
begin
  DECODE ICB(5) =>
  begin
    "3" := begin
      DECODE Z =>
      begin
        DECODE Z =>
        begin
          Q := F * S + Cn;
          I := F * (S + R) + Cn;
        end next
      DECODE mss =>
      begin
        Q := (ASHT @ S100 + S103 @ F); WRITE = Q;
        I := (S103 + R12); I MSS
        DECODE ICB(5) =>
        begin
          Q := ASHT @ S100 + Cn4 @ F;
          I := ASHT @ S100 + (F(3) xpr P.OVR()) @ F
        end
      end
    end
    "4" := begin
      DECODE ICB(5) =>
      begin
        I := ASHT @ F + (S + R) + Cn;
        I := (DECODE Z =>
        begin
          Q := ASHT @ F + S + Cn;
          I := ASHT @ F + (not S) + Cn
        end next
        IF mss => ASHT(3) = S(3) xpr F(3));
        S100 parity: Q100 = Q100 + R12; WRITE = Q
      end
    end
    "6" := begin
      DECODE Z =>
      begin
        DECODE Z =>
        begin
          Q := F * S + Cn;
          I := F * (not S) + Cn
        end next
      DECODE mss =>
      begin
        Q := ASHT @ S100 + S103 @ F;
        I := (S103 + R12);
        ASHT @ S100 = (F(3) xpr P.OVR()) @ F
      end
    end
    Q @ Q100 = Q100 @ Q; WRITE = Q
  end
  "8" := begin
    DECODE ICB(5) =>
    begin
      Q := (ASHT @ F; S103 = F(3); S100 = R12);
      I := (DECODE mss =>
      begin
        B := S103 @ ASHT + F @ S100;
        I := S103 @ ASHT
        + (R(3) xpr F(3)) @ F(2); @ S100
      end
      end
    end
    Q103 @ Q = Q @ Q100; WRITE = B
  end
  "C" := begin
    DECODE Z =>
    begin
      DECODE Z =>
      begin
        Q := F * (S + R) + Cn;
        I := F * ((S + R) - 1) + Cn
      end next
    DECODE ICB(5) =>
    begin
      DECODE mss =>
      begin
        Q := S103 @ ASHT + F @ S100;
        I := S103 @ ASHT
        + (R(3) xpr F(3)) @ F(2); @ S100
      end
    end
    I := (ASHT @ F; S103 = F(3); S100 = R12)
  end
  Q103 @ Q = Q @ Q100; WRITE = Q
end
end
```

Service Facilities[us]

```
I(1) :=
begin
  DECODE ICB(5) =>
  begin
    "0" := DECODE ICB(5) =>
    begin
      DECODE ICB(5) =>
      begin
        ["0":"3","0":"7"] := IF not LSSL => Z = Q(3);
        "4" := Z = ASHT eq1 0;
        "5" := IF mss => Z = S(3);
        "6":"9" := Z = 0 eq1 0;
        "A":"B" := Z = (Q eq1 0) and (F eq1 3);
        "C":"F" := IF mss => Z = R(3) eq1 F(3);
      end
    end
    "1":"F" := Z = ASHT eq1 0
  end
  end
  g1(1) :=
  begin
    DECODE ICB(5) =>
    begin
      DECODE ICB(5) =>
      begin
        "0" := DECODE ICB(5) =>
        begin
          DECODE ICB(5) =>
          begin
            "0":"3" := DECODE Z =>
            begin
              Q := g1 + B;
              I := g1 + R and S
            end
            "4" := DECODE LSSL =>
            begin
              Q := g1 + '000 @ S(3);
              I := g1 = 0
            end
            "5":"8":"9" := g1 = B;
            "6":"F" := DECODE Z =>
            begin
              Q := g1 = 0;
              I := g1 = (not R) and S
            end
            "C":"F" := DECODE Z =>
            begin
              R := g1 + F and S;
              I := g1 + (not R) and S
            end
          end
          I := g1 = 0
        end
        end
        ["1":"8":"9"] := g1 = (not R) and S;
        ["2":"9"] := g1 = R and (not S);
        ["8":"A":"C":"E"] := g1 = R and S;
        ["4":"8"] := g1 = 0;
        ["0":"F"] := g1 = (not R) and (not S)
      end
    end
  end
  p1(1) :=
  begin
    DECODE ICB(5) =>
    begin
      DECODE ICB(5) =>
      begin
        "0" := DECODE ICB(5) =>
        begin
          DECODE ICB(5) =>
          begin
            "0":"3" := DECODE Z =>
            begin
              Q := g1 + S;
              I := g1 + R and S
            end
            "4" := DECODE LSSL =>
            begin
              Q := g1 + S(3); @ S;
              I := g1 = S
            end
            "5" := DECODE Z =>
            begin
              Q := g1 + S;
              I := g1 + not S
            end
            "6":"7" := DECODE Z =>
            begin
              Q := g1 + S;
              I := g1 = (not R) or S
            end
            "8":"9" := g1 + S;
            "C":"F" := DECODE Z =>
            begin
              Q := g1 + F or S;
              I := g1 + (not R) or S
            end
          end
        end
      end
    end
  end
```

••Service Facilities••(urb)

```

H)O :=                                ! Compute zero output

```

[illegible]

APPENDIX 2 (right) AM2910 ISP DESCRIPTION

end 1 end of AM2310 description

A PDP-8 Implemented from AMD Bit-Sliced Microprocessors

Michael Tsao

An example of a microprogrammable system based on the Am2910 sequencer and the Am2901 ALU will illustrate design with bit slices. The target machine is the PDP-8 ISP (see Appendix 1 of Chap. 8). This register-transfer (RT) level design of the micromachine is thus optimized toward the basic PDP-8. However, the general principles involved in microprogramming bit slices are illustrated by this example. A major goal of this design is the clarity of implementation, rather than the economy of design.

Overview

The basic implementation is a *one-stage pipeline* as shown in Fig. 1 in Chap. 13. In this micromachine, the pipeline register stores the current microinstruction, which is being executed by the Am2910 Sequencer and the Am2901 ALU. The status information (zero, overflow, etc.) of the ALU operations is stored in the Status Register. In a one-stage pipeline design, conditional branches can be executed only by the microinstruction following the microcycle that has generated the branching status. The Am2910 sequencer is used instead of the Am2909 to simplify the design and to aid understandability. A more cost-effective design might actually result from using the Am2909 sequencer, since the number of microinstruction types used to emulate the PDP-8 is small. The Am2901 ALU is used because it more closely reflects the ISP of the PDP-8.

A timing diagram for a typical microcycle is shown in Fig. 1. The indicated delays are typical values, illustrating the timing requirements rather than actual component performances. On the rising edge of the system clock, the Pipeline Register latches the microinstruction to be executed during this microcycle. The output of the Pipeline Register is valid 15 ns later. After another 15-ns delay, the Condition Code input to the Am2910 is valid. The microsequencer generates the next microaddress based on the current microinstruction and the Condition Code input. When the microprogram memory output is valid (approximately 130 ns after the rising clock edge), the microcycle can be restarted. Concurrently with the sequencer operation and microword fetch, the Am2901 ALU executes the operations specified by the microword in the Pipeline Register. The output of the ALU is

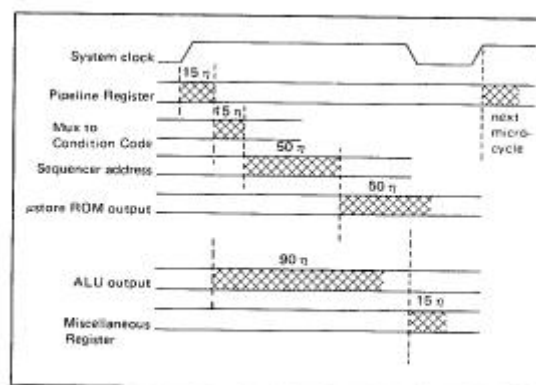


Fig. 1. One-stage pipeline microcycle timing waveform.

valid prior to the falling edge of the system clock. External registers, such as the Memory Address Register (MAR) and the Status Register, use the falling clock edge to latch results from the ALU output port. In this design, the duty cycle of the system clock does not need to be symmetrical at 50 percent.

RT-Level Implementation and the Microword Format

The RT-level implementation of the Am2900/PDP-8 is shown in Fig. 2 for the control part, and in Fig. 3 for the data part. The design can best be explained in conjunction with the microword format shown in Table 1. The ISPS description of the RT-level design is listed in Appendix 2. The following subsections discuss the meaning of each microword field and the associated RT-level components. For each microword field, there are three possible bit sizes: the number of bits *normally* required for the associated components, the *minimum* required for this PDP-8 application, and the *actual* field size used. The position of each field in the microword is defined in the ISPS description. The reason for inserting extra bits is to align the fields on octal boundaries, thus aiding the reading of the encoded microprogram.

Sequencer Instruction and Address Field

The Am2910 sequencer normally requires a 4-bit-wide instruction and a 12-bit-wide "next address" direct input. The microprogram occupies less than 128 words, requiring only 7 bits of address. Two extra instruction bits and two extra address bits are inserted as 0s in this design example for octal boundary alignment.

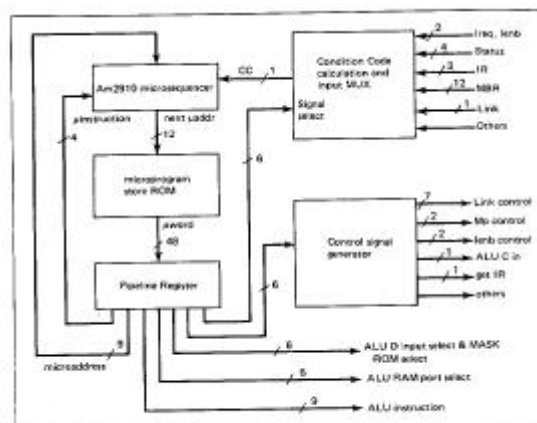


Fig. 2. Micromachine control and sequencer.

Out of sixteen Am2910 instructions, only 4 are used in this example: Conditional Jump Subroutine (CJS, #01), Conditional Jump (CJP, #03), Conditional Return from Subroutine (CRTN, #12), and Continue (CONT, #16). Therefore, **it** is theoretically possible to use only 2 bits of information to specify these four actions.

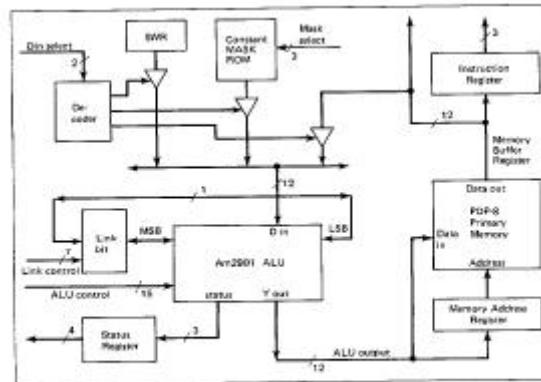


Fig. 3. Micromachine—ALU and data.

Table 1 The Microword Format and Required Bits per Field

<i>Bits per field</i>	<i>Normal</i>	<i>Minimum</i>	<i>Actual (ISP)</i>
Micro sequencer control			
Microinstruction	4	2	6
Next microaddress	12	7	9
Condition code select	(6)	6	6
ALU control			
ALU instruction			
Source	3	3	3
Function	3	3	3
Destination	3	3	3
RAM A port select	4	3	3
RAM B port select	4	3	3
Direct input select	(2)	2	3
Constant mask select	(3)	3	3
Miscellaneous control signals			

Control signal select	(4)	4	6
Total	48	39	48

Condition Code Input Selection

There is only one condition code (CC) input for the Am2910. The status conditions have to be multiplexed into this input. The assignments for the multiplexer input lines can be found in the ISP description in Appendix 1 (ISPS procedure Condition .Code). Five bits are used to select one out of 32 different input signals. The sixth bit in this field is used to select between the original signal and the complement of the signal. In this manner, the micromachine can branch when the signal is either high or low. When an unconditional microprogram branch is required, a logic 0 can be selected for the CC input.

Each bit from the Instruction Register (IR, 5 bits) or from the Memory Buffer Register (MBR, 12 bits) can be selected individually. This capability is used for the basic PDP-8 instruction decode, effective address calculation, and the Group 7 microinstruction decode. Random combinational logic is used to generate a single skip enable signal for the portion of the microprogram that decodes the PDP-8 skip conditions. Interrupt requests are also handled by using combinational logic in a similar manner.

ALU Operations and the Link Bit

Three Am2901 ALU chips are cascaded to form the PDP-8 ALU section. The ALU requires a 9-bit opcode: source, function, and destination. Six bits are used to encode the A port (3 bits) and B port (3 bits) select, since only a subset of the sixteen ALU RAM registers is used in this implementation.

The PDP-8 Link bit is constructed from random logic controlled by a set of signals. For economic reasons, random logic is used rather than adding another Am2901 chip. The Link bit does not correspond to any Am2901 function, and its control would

have to be separately microprogrammed. Another alternative for the PDP-8 Link bit is to use one of the Am2901 RAM registers for storing the value. In this case, additional Link-handling microcode would have to be inserted after each PDP-8 ALU operation, increasing the target instruction execution time.

Data Input to the ALU

There is only one method of writing external data into the Am2901 ALU. It is through the Direct (D) input. In this PDP-8 design, three sources are connected to share the D input: data from the main memory (MBR), constants for ALU operations (the Mask ROM), and data in the switch register (SWITCHES). These three sources are connected by an input bus to the D input port on the ALU. The microword selects which one of the three will be the source during any given microcycle.

The use of a separate ROM to store the constants can be debated. An alternative is to store the constants in the microword. It is wasteful to dedicate a microword bit field to this purpose, since the width of this field must be the same as the ALU width and constants are used

infrequently. If the microword fields are multiplexed, we violate the design goal of clarity. Hence, a constant ROM is a good compromise between the two conflicting objectives. One need only store the address of the constant in the microprogram.

Miscellaneous Control Signals

The data part of this design requires many miscellaneous control signals. For example, the Link bit uses seven different signals to control its operation. Analysis indicates that only one of these signals needs to be asserted during any given microcycle. The Miscellaneous Control Select field in the microword selects one and only one signal during each microcycle. The selection code is decoded and directed to the associated destinations. The assignments of the signals can be found in the ISPS description.

The PDP-8 Primary Memory

The primary memory (MP) for the PDP-8 target machine is assumed to be constructed from "static" semiconductor memory chips. In this type of memory, the output constantly displays the content of the location selected by the address input, unless a write operation is in progress. In this PDP-8 design, the ALU output is connected with the Memory Address Register (MAR) and with the data input port of the MP. When the write enable line of the MP is asserted, the content of the ALU output port is latched into the location selected by the MAR. The Memory Buffer Register (MBR), an ISP implementation pseudoregister, is constantly displaying the content of the location selected by the MAR. For the ISPS simulation, the memory access speed is assumed to be less than one microcycle. One can read the value of MBR (containing data from MP) two microcycles after a "write" into the MAR.

The Microprogram

The encoded microprogram that emulates the PDP-8 basic instruction set is listed in Appendix 2. This program listing is extracted from an ISPS simulator command file used to simulate this microprogrammable machine. The content of the constant ROM (Mask) is defined using the ISPS simulator "set" command, e.g., "set Mask[4] = #0177." The content of the microprogram store is also defined in this manner. As an example, the instruction fetch cycle is now described. (For readability, the encoded microword is broken into seven fields separated by dashes.)

```
set uMP[000] = #03-010-10-403-12-00-10
```

```
!RUN: MAR ← LastPC ← PC, IF PDP8.go = 0 goto HALT:
```

If the PDP-8. go bit is off (Condition code select 10), the microprogram jumps to Halt: (location 010). The content of PC (ALU RAM[1]) is pushed to the ALU output. The value is also latched into LastPC (ALU RAM[2]). Concurrently, the value is latched into the Memory Address Register (MAR) using the control code 10.

```
set uMP[001] = # 16-000-00-503-11-21-00 !PC ← PC+1
```

The value #0001 is selected from the constant Mask ROM (21). The PC value is selected at the ALU A port, added to the constant, and then latched back into PC.

```
set uMP[002] = #03-040-41-703-05-10-15
```

!IR← ALU.Mb← MBR, goto Exec:

The content of the Memory Buffer Register (obtained by the MP[MAR] operation) is latched into the ALU. Mb (ALU RAM[5]). In this cycle, the MBR is also latched into the Instruction Register (IR) by the control signal 15. The micro program jumps to the instruction execution section (location 040, Exec:) by forcing a pass-test condition (41) into the Am2910 sequencer Condition Code input.

set uMP[004] = #03-000-03-741-00-20-10

!ENDex: MAR← 0, IF no interrupt goto RUN:

When the instruction execution is finished, the microprogram returns to this point. The MAR is set to zero in anticipation of interrupt servicing. The MAR will be reset to the correct PC value by microinstruction uMP[001] later on. If the interrupt request is not granted (condition code 03), the microprogram jumps back to RUN: (location 000). Otherwise, the program continues to location uMP[005] to handle the interrupt.

Implementation and Simulation Results

The micromachine and the microcode were simulated and tested by the ISPS simulator. The results are presented here.

Chip Count

Since the micromachine was not actually built, the chip count is an estimate of the required hardware parts. The goal of this exercise is to identify the inefficient area in terms of the parts count, and to suggest alternative IC chip types that may reduce the parts count. (See Table 2.)

The parts count for this microprogrammed PDP-8 implementation is 35 chips. Of these IC parts, over two-thirds (25 chips) are SSI or MSI types. If IC custom-made parts are available for the Link bit, the Skip-condition generate, and the Pipeline Register, the design can be reduced to 22 chips.

Target-Machine Instruction Execution Speed

Two methods of comparing this microprogrammed PDP-8 and a basic PDP-8 are discussed here. By counting the average number of microinstructions executed for a target instruction, one can estimate the execution speed of the emulated PDP-8. Or one can compare the execution speed of the two LSPS simulators.

Table 2 Chip Count for a Microprogrammed POP-B

<i>Chip count</i>	<i>Description</i>
6	Microstore. The microword width is between 39 bits and 48 bits (see Table 1). In using 8-bit-wide ROM or EPROM parts, six such chips are required. Since the microprogram is less than 128 words (7 address bits), many commercially available memory chips can be used here.

- 6 Pipeline Register (Pipe). Eight-bit-wide *D* flip-flops are assumed here. This register is very expansive in terms of chip count. An alternative would be having a special ROM type that can latch the data in the output buffer. Another alternative is to latch the microaddress instead of the microword. In this second design, the microword fetch and ALU-Sequencer operations are in series rather than in parallel as in the original design. This is a classical cost- performance tradeoff.
- 1 Am2910 microsequencer. The advantage of using the Am2910 instead of the Am2909 Sequencer is evident here. The Am2909 requires two chips instead of one Am2910 for this example.
- 3 Am2901 ALU bit slices. Three slices are used to provide the 12-bit-wide PDP-8 data path.
- 5 (estimated) Link bit and associated hardware. The link bit in this design is constructed of a *D* flip-flop, some tristate drivers, and input multiplexers. SSI implementation of the Link bit requires 14 percent (5 out of 35) of the total chip count. An alternative is to use a custom-made MSI chip for the Link bit. A second alternative is to implement the Link bit in the ALU RAM registers. In this second design, additional microcode will have to be inserted to handle the special cases, degrading the overall performance.
- 3 Condition Code input multiplexer. Two 16-to-1 MUXs and two 2-to-1 MUXs.
- 4 PDP-8 Skip condition generate. The argument for a custom MSI chip can also be made here.
- 3 Constant Mask ROM and associated ALU *D* input selection control. The Constant Mask uses two ROM chips. The *D* input control uses one 2-to-4 decoder. The source registers for the ALU *D* input bus are assumed to have build-in tristate drivers.
- 4 (estimated) Other miscellaneous parts.

For each target PDP-8 instruction, the microprogram must execute the following number of microinstructions (Table 3).

On the average, 18 microwords ($4 + 3 + 6 + 5$ or $4 + 3 + 11$) are needed to do one PDP-8 target instruction. At the manufacturer-recommended microcycle time of 150 ns, and not counting the PDP-8 Mp access time, the microprogram execution speed is 2.7 μ s per target instruction ($150 \text{ ns} \times 18$). The Mp access time is usually quoted at 1.3 μ s for PDP-8/E and /M [Bell, Mudge, and McNamara, 1978]. For an average instruction (i.e., indirect memory reference), three memory accesses are required: instruction fetch, pointer to data (one level of indirection), and the actual data fetch. When these are added to the 2.7- μ s microprogram execution time, the projected maximum average instruction time is 6.6 μ s.

Another method of comparison involves the ISPS simulator. Several PDP-8 benchmark and diagnostic programs were simulated. The CPU times used by each simulator were compared. The microcoded PDP-8 uses approximately 20 times the CPU time used by the basic PDP-8 ISP. Translated into simulation CPU time, the ISP simulator of the micromachine executes approximately 1.5 PDP-8 target instructions for every CPU second on a DEC KL-10 processor.

Table 3 Average Number of Microinstructions Executed for a Target Instruction

Words Description

- | | |
|---|---|
| 4 | PDP-8 instruction fetch cycle. Check PDP-8.go, fetch target instruction, increment PC, check interrupt conditions. |
| 3 | Instruction decodes. A straightforward binary decision decode tree is implemented in microcode. An alternative is to use the Instruction Decode Mapping ROM capability of the Am2910. The advantage of this alternative is not clear in view of the simple PDP-8 ISP. |
| 6 | Effective Address Calculation. Depending on the addressing mode, there are five possibilities |

2 words PDP-8 Page 0 address
4 words current page
6 words indirect address, Page 0
8 words indirect address, current page
9 words auto index

On the average, approximately six microinstructions are needed to calculate the PDP-8 effective address (equivalent to the Page 0 indirect address).

- | | |
|------|---|
| 5 | Memory Reference Instructions. For each target instruction, the microcode fetches data from primary memory, executes the operation, and deposits the result in memory. Depending on the particular target instruction, anywhere between two microinstructions (JMP) and eight microinstructions (ISZ) are needed. On the average, five microinstructions are assumed. |
| (11) | PDP-8 OPR group microinstructions. The decoding and execution of the PDP-8 OPR instructions are highly sequential in nature. Therefore, 11 microinstructions executed is taken as the average. |

Summary

In this chapter, the design of a microprogrammed PDP-8 was presented. The central component of this micromachine was the AMD bit-sliced microprocessor. Although the design was optimized toward the basic PDP-8 configuration, many issues common to all microprogramming and RT-level hardware designs were illustrated. In simulating the micromachine, the usefulness of the ISP descriptive language as a design tool was also demonstrated.

References

Bell, Mudge, and McNamara [1978]

APPENDIX 1 ISP OF A PDP-8 EMULATOR USING THE AM2901 AND AM2910

```

AM29 :=
begin
  ! ISP of a PDP8 emulator using the AM2901 and AM2910 bit slice uP chips
  ! The AM2901 description is expanded to fit the 12 bits wide PDP8 data path.
  ! The AM2910 sequencer uses 7 address bits to address 128 microwords.

  ! This AM29 description contains the following major sections.
  ! (1) declarations for the PDP8 target machine, the sequencer, and the ALU.
  ! (2) implementation pseudo registers for the ISP description.
  ! (3) the main microcycle execution loop.
  ! (4) the sequencer Condition Code (CC) input multiplexer.
  ! (5) the generator for the PDP8 skip condition and miscellaneous control signals.

  **PDP8.State**
  MP[0:4095] <= 11:0;      ! Basic PDP8 4k memory
  LDR[0:15] <= 11:0;      ! Link Register
  Interrupt.enable <= 1;   ! Interrupt enable
  PDP8.go <= 1;           ! Run bit for the PDP8 target machine

  INCR[4:0] <= 11:0;      ! Instruction register & ps & ls
  ps <= 11:0;             ! Page bit
  ls <= 11:0;             ! Redirect bit
  MAR[11:0] <= 11:0;      ! Memory Address Register

  **Sequencer.State**
  uMP[0:127] <= 47:0;     ! Microprogram memory
  PIPE[47:0] <= 11:0;     ! Pipeline Register

  ! sequencer controls
  uSrc[0] <= PIPE[47:40]; ! AM2910 instruction, padded to 6 bits
  uNextAddr[0:0] <= PIPE[43:33]; ! next micro-address field
  CCsel[0:0] <= PIPE[32:27]; ! select condition code input

  ! ALU controls
  ALUir[0:0] <= PIPE[26:18]; ! ALU instruction
  irsrc[0:0] <= ALUir[0:0]; ! Source
  irfnc[0:0] <= ALUir[0:0]; ! Function
  irdst[0:0] <= ALUir[0:0]; ! Destination
  ALUport[0:0] <= PIPE[17:12]; ! ALU RAM port select
  DirctSel[0:0] <= PIPE[11:0]; ! ALU direct input select
  MaskSel[0:0] <= PIPE[0:0]; ! Constant mask select

  MiscCtrl[0:0] <= PIPE[0:0]; ! Miscellaneous Control signals

  **ALU.State**
  mask[0:15] <= 11:0;     ! constant mask ROM
  status[0:0] <= 11:0;    ! ALU result status
  SEQ.overflow <= status[0]; ! SEQ stack overflow
  ALU.overflow <= status[0]; ! ALU result overflow
  ALU.negative <= status[1]; ! ALU result is negative
  ALU.zero <= status[0]; ! ALU result is zero

  **Implementation Registers**
  AM29.go <= 1;           ! go bit for the micro machine

  Interrupt.request[0]; ! Interrupt request
  MBR[0:15] <= MBR[0:15]; ! Memory Buffer Register, output of MBR[MAR]
  group <= MBR[0];        ! PDP8 bit assignments
  CLAC <= MBR[1];        ! Microinstruction group
  CLAC <= MBR[1];        ! Clear AC
  CLAC <= MBR[1];        ! Clear Link
  CMAC <= MBR[2];        ! Complement AC
  CMAC <= MBR[2];        ! Complement Link
  RALC <= MBR[3];        ! Rotate right
  RALC <= MBR[3];        ! Rotate left
  RTAC <= MBR[4];        ! Rotate twice
  RTAC <= MBR[4];        ! Rotate twice
  IAC <= MBR[5];        ! Increment AC
  IAC <= MBR[5];        ! Increment Link
  SMC <= MBR[6];        ! Skip on minus AC
  SMC <= MBR[6];        ! Skip on minus Link
  SPC <= MBR[7];        ! Skip on positive AC
  SPC <= MBR[7];        ! Skip on positive Link
  SZAC <= MBR[8];        ! Skip on zero AC
  SZAC <= MBR[8];        ! Skip on zero Link
  SMC <= MBR[9];        ! Skip on AC not zero
  SMC <= MBR[9];        ! Skip on Link not zero
  SZL <= MBR[10];        ! Skip on Link zero
  SZL <= MBR[10];        ! Skip on Link zero
  LSAC <= MBR[11];        ! Logical or AC with switches
  LSAC <= MBR[11];        ! Logical or Link with switches
  RLTC <= MBR[12];        ! Halt the processor

  uMP.out <= 47:0;       ! uStore output
  CCcode <= 11:0;        ! condition code input
  DirctIn <= 11:0;       ! direct input to ALU
  ALU.Cin <= 11:0;       ! ALU carry in

  Temp.AM2910[15:0] <= 11:0; ! effective uStore address
  uMP.addr[0:0] <= Temp.AM2910[0:0]; ! effective uStore address

  Temp.AM2910[17:0] <= 11:0; ! ALU result Y output
  ALU.out[0:0] <= Temp.AM2910[17:0]; ! ALU carry out
  ALU.LSB <= Temp.AM2910[16]; ! ALU LSB output from RAM shifter
  ALU.MSB <= Temp.AM2910[17]; ! ALU MSB output from RAM shifter

  **AM29.Execution**
  start.AM29[0:0] <= 11:0;
  begin
    ! initialize the micro machine
    AM29.go <= 1; ! get the uMachine going
    uMP.out <= uMP[0]; ! start at uAddr 0

    ! initialize the target machine
    ! force interrupt handling which begins at PDP8 PC=1
    PDP8.go <= 1; ! get the target machine going
    Interrupt.enable <= 1; ! enable interrupt
    Interrupt.request <= 1; ! request interrupt

    run.AM29 <= 1; ! use uCycle
    begin
      ! first half of the cycle
      PIPE <= uMP.out.next ! latch uWord

      uALU.opr <= 11:0; ! ALU operations
      begin
        DECODE DirctSel <= 1; ! select Direct input to ALU
        begin
          #0 <= Din <= 1;
          #1 <= Din <= MBR;
          #2 <= Din <= mask[MaskSel];
          #3 <= Din <= switches;
        end;
        DECODE MiscCtrl <= 1; ! set ALU carry in bit
        begin
          #0 <= ALU.Cin <= 1;
          OTHERWISE <= ALU.Cin <= 0;
        end;
      end;
      ! Do ALU computation, input instruction, uAddr, uPort.
      ! Direct input, carry in, MSD, LSD, and enable output
      Temp.AM2910 <= AM2910[ALUir]; ! 3 B ALUport[0:33];
      ! 6 B ALUport[0:33]; Dir, ALU.Cin, L, L, 'S'
      end; ! and of uALU.opr

      uSEQ.opr <= 11:0; ! sequencer operation
      begin
        Condition.Code <= 11:0; ! next
      end;
      ! Do sequencer computation, input instruction, next address,
      ! condition code, etc.
      Temp.AM2910 <= AM2910[uMP[0:0]; ! '0' B uNextAddr[0:0];
      ! CCcode, '2' B uMP.out <= uMP[uMP.addr]; ! uStore ROM access
      end; ! end of uSEQ.opr

      ! Second half of the cycle
      status[0:0] <= Temp.AM2910[15]; ! Temp.AM2910[14];
      ! Temp.AM2910[15]; ! Temp.AM2910[12];
      ! Do Control <= next
      ! If AM29.go <= 1; RESTART run.AM29;
      end; ! end of start.AM29

    end; ! end of run.AM29

  **Condition.Code**
  Condition.Code <= 11:0; ! calculate condition code
  begin
    Decode { '2 B CCsel[0:0] } <= 1; ! look at lower 2 bits
    begin
      #0 <= CCcode <= 1; ! #0 <= always pass test
      #1 <= CCcode <= 1; ! #1 <= always fail test
      #2 <= CCcode <= 1; ! #2 <= always pass test
      #3 <= CCcode <= 1; ! #3 <= always fail test
      #4 <= CCcode <= 1; ! calculate skip condition
      #5 <= CCcode <= 1; ! Interrupt.request AND Interrupt.enable
      #6 <= CCcode <= 1; ! check for interrupt handling
      #7 <= CCcode <= 1; ! ALU.x, ALU output eq? 0
      #8 <= CCcode <= 1; ! ALU.n, ALU MSB
      #9 <= CCcode <= 1; ! ALU.overflow, ALU operation overflow
      #10 <= CCcode <= 1; ! SEQ.overflow, sequencer stack overflow
      #11 <= CCcode <= 1; ! target machine RSN bit
      #12 <= CCcode <= 1; ! LSR of the 3 bits IR
      #13 <= CCcode <= 1; ! MSB of the 3 bits IR
      #14 <= CCcode <= 1; ! page bit
      #15 <= CCcode <= 1; ! indirect bit
    end;
  end;
end;

```

```

#20 := CCode = MRC(0);      ! JAC, MRC(0), LSR
#21 := CCode = MRC(0);      ! RTE, ALT
#22 := CCode = MRC(0);      ! HSE, OSH
#23 := CCode = MRC(0);      ! BAE, Is
#24 := CCode = MRC(0);      ! CHL, SNL, SZL
#25 := CCode = MRC(0);      ! CHA, SEA, SRA
#26 := CCode = MRC(0);      ! CLL, SRA, SRA
#27 := CCode = MRC(0);      ! CLA
#28 := CCode = MRC(0);      ! group
#29 := CCode = MRC(0);      !
#30 := CCode = MRC(0);      !
#31 := CCode = MRC(0);      !
#32 := CCode = MRC(0);      !
#33 := CCode = MRC(0);      ! MRC(0), HSE
#34 := CCode = L;          ! Link bit

OTHERWISE := CCode = 0
end mrc

! when CCode=0 will pass test if the selected signal is low, "0"
! when CCode=1 will pass test if the selected signal is high, "1"
if CCode=0 then CCode = not CCode
end.

**Skip Conditions for PDP8**
Skip.Cond :=
begin
  DECODE is >
  begin
    #0 := Skip.Cond =
      (SNL and L) or
      (SZA and ALD.x) or
      (SRA and ALD.x);
    #1 := Skip.Cond =
      (NOT (SZL or SAA or SPA)) or
      (SZL and NOT L) or
      (SRA and NOT ALD.x) or
      (SPA and NOT ALD.x);
  end
end.

**Miscellaneous Controls**
Do.Control :=
begin
  DECODE MISCtr >
  begin
    #0 := MD.OP();          ! no op
    #1 := IF ALU.Cout >
      L = not L;
    #2 := L = ALD.Lab;      ! L = old LSR
    #3 := L = ALD.MSB;      ! L = old MSR
    #4 := L = 0;           ! clear Link
    #5 := IF CLL > L = 0;   ! conditional clear
    #6 := L = not L;        ! negate Link
    #7 := IF CHL > L = not L; ! conditional negate
    #8 := (MAR = ALU.out.next) ! set MAR
    #9 := (MAR = MP(MAR));    ! read PDP8 main memory
    #10 := (MP(MAR) = ALU.out.next) ! write into PDP8 MP
    #11 := (MAR = MP(MAR));
    #12 := Interrupt.enable = 0; ! clear Interrupt enable
    #13 := Interrupt.enable = 1; ! set Interrupt enable
    #14 := ALU.Cin = 1;       ! set ALU carry in
    #15 := IR = MRC(0-4);    ! get instruction
    #16 := PDP8.go = 0;      ! stop the target machine
    #17 := AMB8.go = 0;      ! stop the micro-machine

    OTHERWISE := MD.OP();
  end
end.
! logical end of the AMB8 description

```

APPENDIX 2 SIMULATOR COMMAND FILE FOR AM2900 IMPLEMENTATION OF THE PDP-8

```

; Simulator command file for the AM2900 implementation of the PDP-8
;
; (radix octal)
;
; constant Mask ROM, used as input to the ALU Direct input port
; Mask[0]=#0000
; Mask[1]=#0001
; Mask[2]=#0002
; Mask[3]=#0004
; Mask[4]=#0008
; Mask[5]=#0010
; Mask[6]=#0020
; Mask[7]=#0040
; Mask[8]=#0080
; Mask[9]=#0100
; Mask[10]=#0200
; Mask[11]=#0400
; Mask[12]=#0800
; Mask[13]=#1000
; Mask[14]=#2000
; Mask[15]=#4000
; Mask[16]=#8000
; Mask[17]=#0000
; Mask[18]=#0000
; Mask[19]=#0000
; Mask[20]=#0000
; Mask[21]=#0000
; Mask[22]=#0000
; Mask[23]=#0000
; Mask[24]=#0000
; Mask[25]=#0000
; Mask[26]=#0000
; Mask[27]=#0000
; Mask[28]=#0000
; Mask[29]=#0000
; Mask[30]=#0000
; Mask[31]=#0000
;
; the micro program
; micro word format
; #| 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
; #| 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
; #| 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
; #| 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
; #| 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
; #| 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
; #| 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
; #| 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
; #| 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
; #| 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
; #| 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
; #| 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
; #| 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
; #| 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
; #| 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
; #| 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
; #| 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
; #| 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271
; #| 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287
; #| 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303
; #| 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319
; #| 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335
; #| 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351
; #| 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367
; #| 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383
; #| 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399
; #| 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415
; #| 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431
; #| 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447
; #| 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463
; #| 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479
; #| 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495
; #| 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511
; #| 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527
; #| 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543
; #| 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559
; #| 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575
; #| 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591
; #| 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607
; #| 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623
; #| 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639
; #| 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655
; #| 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671
; #| 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687
; #| 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703
; #| 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719
; #| 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735
; #| 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751
; #| 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767
; #| 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783
; #| 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799
; #| 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815
; #| 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831
; #| 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847
; #| 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863
; #| 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879
; #| 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895
; #| 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911
; #| 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927
; #| 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943
; #| 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959
; #| 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975
; #| 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991
; #| 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007
; #| 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023
; #| 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039
; #| 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055
; #| 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071
; #| 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087
; #| 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103
; #| 1104 1105 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119
; #| 1120 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135
; #| 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150 1151
; #| 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165 1166 1167
; #| 1168 1169 1170 1171 1172 1173 1174 1175 1176 1177 1178 1179 1180 1181 1182 1183
; #| 1184 1185 1186 1187 1188 1189 1190 1191 1192 1193 1194 1195 1196 1197 1198 1199
; #| 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215
; #| 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231
; #| 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247
; #| 1248 1249 1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263
; #| 1264 1265 1266 1267 1268 1269 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279
; #| 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1293 1294 1295
; #| 1296 1297 1298 1299 1300 1301 1302 1303 1304 1305 1306 1307 1308 1309 1310 1311
; #| 1312 1313 1314 1315 1316 1317 1318 1319 1320 1321 1322 1323 1324 1325 1326 1327
; #| 1328 1329 1330 1331 1332 1333 1334 1335 1336 1337 1338 1339 1340 1341 1342 1343
; #| 1344 1345 1346 1347 1348 1349 1350 1351 1352 1353 1354 1355 1356 1357 1358 1359
; #| 1360 1361 1362 1363 1364 1365 1366 1367 1368 1369 1370 1371 1372 1373 1374 1375
; #| 1376 1377 1378 1379 1380 1381 1382 1383 1384 1385 1386 1387 1388 1389 1390 1391
; #| 1392 1393 1394 1395 1396 1397 1398 1399 1400 1401 1402 1403 1404 1405 1406 1407
; #| 1408 1409 1410 1411 1412 1413 1414 1415 1416 1417 1418 1419 1420 1421 1422 1423
; #| 1424 1425 1426 1427 1428 1429 1430 1431 1432 1433 1434 1435 1436 1437 1438 1439
; #| 1440 1441 1442 1443 1444 1445 1446 1447 1448 1449 1450 1451 1452 1453 1454 1455
; #| 1456 1457 1458 1459 1460 1461 1462 1463 1464 1465 1466 1467 1468 1469 1470 1471
; #| 1472 1473 1474 1475 1476 1477 1478 1479 1480 1481 1482 1483 1484 1485 1486 1487
; #| 1488 1489 1490 1491 1492 1493 1494 1495 1496 1497 1498 1499 1500 1501 1502 1503
; #| 1504 1505 1506 1507 1508 1509 1510 1511 1512 1513 1514 1515 1516 1517 1518 1519
; #| 1520 1521 1522 1523 1524 1525 1526 1527 1528 1529 1530 1531 1532 1533 1534 1535
; #| 1536 1537 1538 1539 1540 1541 1542 1543 1544 1545 1546 1547 1548 1549 1550 1551
; #| 1552 1553 1554 1555 1556 1557 1558 1559 1560 1561 1562 1563 1564 1565 1566 1567
; #| 1568 1569 1570 1571 1572 1573 1574 1575 1576 1577 1578 1579 1580 1581 1582 1583
; #| 1584 1585 1586 1587 1588 1589 1590 1591 1592 1593 1594 1595 1596 1597 1598 1599
; #| 1600 1601 1602 1603 1604 1605 1606 1607 1608 1609 1610 1611 1612 1613 1614 1615
; #| 1616 1617 1618 1619 1620 1621 1622 1623 1624 1625 1626 1627 1628 1629 1630 1631
; #| 1632 1633 1634 1635 1636 1637 1638 1639 1640 1641 1642 1643 1644 1645 1646 1647
; #| 1648 1649 1650 1651 1652 1653 1654 1655 1656 1657 1658 1659 1660 1661 1662 1663
; #| 1664 1665 1666 1667 1668 1669 1670 1671 1672 1673 1674 1675 1676 1677 1678 1679
; #| 1680 1681 1682 1683 1684 1685 1686 1687 1688 1689 1690 1691 1692 1693 1694 1695
; #| 1696 1697 1698 1699 1700 1701 1702 1703 1704 1705 1706 1707 1708 1709 1710 1711
; #| 1712 1713 1714 1715 1716 1717 1718 1719 1720 1721 1722 1723 1724 1725 1726 1727
; #| 1728 1729 1730 1731 1732 1733 1734 1735 1736 1737 1738 1739 1740 1741 1742 1743
; #| 1744 1745 1746 1747 1748 1749 1750 1751 1752 1753 1754 1755 1756 1757 1758 1759
; #| 1760 1761 1762 1763 1764 1765 1766 1767 1768 1769 1770 1771 1772 1773 1774 1775
; #| 1776 1777 1778 1779 1780 1781 1782 1783 1784 1785 1786 1787 1788 1789 1790 1791
; #| 1792 1793 1794 1795 1796 1797 1798 1799 1800 1801 1802 1803 1804 1805 1806 1807
; #| 1808 1809 1810 1811 1812 1813 1814 1815 1816 1817 1818 1819 1820 1821 1822 1823
; #| 1824 1825 1826 1827 1828 1829 1830 1831 1832 1833 1834 1835 1836 1837 1838 1839
; #| 1840 1841 1842 1843 1844 1845 1846 1847 1848 1849 1850 1851 1852 1853 1854 1855
; #| 1856 1857 1858 1859 1860 1861 1862 1863 1864 1865 1866 1867 1868 1869 1870 1871
; #| 1872 1873 1874 1875 1876 1877 1878 1879 1880 1881 1882 1883 1884 1885 1886 1887
; #| 1888 1889 1890 1891 1892 1893 1894 1895 1896 1897 1898 1899 1900 1901 1902 1903
; #| 1904 1905 1906 1907 1908 1909 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919
; #| 1920 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935
; #| 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951
; #| 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967
; #| 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983
; #| 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999
; #| 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015
; #| 2016 2017 2018 2019 2020 2021 2022 2023 2024 2025 2026 2027 2028 2029 2030 2031
; #| 2032 2033 2034 2035 2036 2037 2038 2039 2040 2041 2042 2043 2044 2045 2046 2047
; #| 2048 2049 2050 2051 2052 2053 2054 2055 2056 2057 2058 2059 2060 2061 2062 2063
; #| 2064 2065 2066 2067 2068 2069 2070 2071 2072 2073 2074 2075 2076 2077 2078 2079
; #| 2080 2081 2082 2083 2084 2085 2086 2087 2088 2089 2090 2091 2092 2093 2094 2095
; #| 2096 2097 2098 2099 2100 2101 2102 2103 2104 2105 2106 2107 2108 2109 2110 2111
; #| 2112 2113 2114 2115 2116 2117 2118 2119 2120 2121 2122 2123 2124 2125 2126 2127
; #| 2128 2129 2130 2131 2132 2133 2134 2135 2136 2137 2138 2139 2140 2141 2142 2143
; #| 2144 2145 2146 2147 2148 2149 2150 2151 2152 2153 2154 2155 2156 2157 2158 2159
; #| 2160 2161 2162 2163 2164 2165 2166 2167 2168 2169 2170 2171 2172 2173 2174 2175
; #| 2176 2177 2178 2179 2180 2181 2182 2183 2184 2185 2186 2187 2188 2189 2190 2191
; #| 2192 2193 2194 2195 2196 2197 2198 2199 2200 2201 2202 2203 2204 2205 2206 2207
; #| 2208 2209 2210 2211 2212 2213 2214 2215 2216 2217 2218 2219 2220 2221 2222 2223
; #| 2224 2225 2226 2227 2228 2229 2230 2231 2232 2233 2234 2235 2236 2237 2238 2239
; #| 2240 2241 2242 2243 2244 2245 2246 2247 2248 2249 2250 2251 2252 2253 2254 2255
; #| 2256 2257 2258 2259 2260 2261 2262 2263 2264 2265 2266 2267 2268 2269 2270 2271
; #| 2272 2273 2274 2275 2276 2277 2278 2279 2280 2281 2282 2283 2284 2285 2286 2287
; #| 2288 2289 2290 2291 2292 2293 2294 2295 2296 2297 2298 2299 2300 2301 2302 2303
; #| 2304 2305 2306 2307 2308 2309 2310 2311 2312 2313 2314 2315 2316 2317 2318 2319
; #| 2320 2321 2322 2323 2324 2325 2326 2327 2328 2329 2330 2331 2332 2333 2334 2335
; #| 2336 2337 2338 2339 2340 2341 2342 2343 2344 2345 2346 2347 2348 2349 2350 2351
; #| 2352 2353 2354 2355 2356 2357 2358 2359 2360 2361 2362 2363 2364 2365 2366 2367
; #| 2368 2369 2370 2371 2372 2373 2374 2375 2376 2377 2378 2379 2380 2381 2382 2383
; #| 2384 2385 2386 2387 2388 2389 2390 2391 2392 2393 2394 2395 2396 2397 2398 2399
; #| 2400 2401 2402 2403 2404 2405 2406 2407 2408 2409 2410 2411 2412 2413 2414 2415
; #| 2416 2417 2418 2419 2420 2421 2422 2423 2424 2425 2426 2427 2428 2429 2430 2431
; #| 2432 2433 2434 2435 2436 2437 2438 2439 2440 2441 2442 2443 2444 2445 2446 2447
; #| 2448 2449 2450 2451 2452 2453 2454 2455 2456 2457 
```