Chapter 34

TMS1000/1200: Chip Architecture and Operation[1]

Introduction

The TMS1000/1200 functional block diagram (Fig. 1) shows all major logic blocks and major data paths in the TMS1000/1200 architecture. The ROM, ROM addressing, and instruction decode are on the left side of the diagram. On the right side of the diagram are the adder/comparator, the RAM, the registers for addressing the RAM, and the accumulator, which is the main working register. The major logic blocks are interconnected to the adder with four-bit parallel data paths. The various portions of the architecture will be discussed in the following paragraphs.

Mp State

The ROM has 8,192 possible matrix points (1,024 eight-bit words) where MOS transistors are placed to define the bit patterns of the machine language code. The ROM is organized into 16 pages of 64 words.

There are four RAM files, each containing 16 four-bit words in the RAM's 256-bit matrix (shown in the upper right of Fig. 1).

There are two modes of RAM access (read and write) during the instruction cycle:

1 Data may be read out of the RAM for the purpose of addition, subtraction, or transfer to the other registers.

2 Data is stored in the RAM via the write bus.

Two sources of information are written into the RAM; these sources are selected by the write multiplexer (shown on the right side of the function diagram, Fig. 1). In one mode the multiplexer selects the accumulator information to be written into the RAM (uses the STO microinstruction). The accumulator data is transferred to memory after data is read from the RAM but before the ALU results are stored into the accumulator. In the second mode, the constant and K-input logic is written into the RAM (by the CKM microinstruction). The constants from the ROM instruction bus are transferred to the RAM directly, and an optional data path from K1, K2, K4, and K8 exists although not selected in the standard instruction set. Four RAM bits are carried on the read bus to either the P-multiplexer or the N-multiplexer and then to the adder/comparator.

[1]Abstracted from *TMS1000 Programmer's Reference Manual*, Texas Instruments, Inc., 1975.

Pc State

a PA<0:3>\Page.Address.Register. Contains the number of the page within the ROM being addressed.

b PB<0:3>\Page.Buffer.Register. The PB is loaded with a new page address which is then shifted into the PA for a successful branch or call. The PB is changed by the load page (LDP) instruction.

c PC<0:5>\Program.Counter. Contains the current location of the word (within the page) being addressed.

d SR<0:5>\Subroutine.Return.Register. Contains the return word address in the call subroutine mode.

e X<0:1>. Designates which of four RAM files are being accessed.

f Y<0:3>. Designates which of 16 four-bit words are being accessed in the specified RAM file.

g R<0:12>. Output register to control external devices.

h O<0:4>. Output register for display.

i K<0:3>. Input register.

j A<0:3>\Accumulator.

k Status.Logic<>. One-bit flag containing the status of previously executed instructions.

On powerup, the program counter is reset to location zero, and the PA is set to 15. Then the program counter counts to the next ROM address in a pseudorandom sequence. The sequence of addresses in the program counter can be altered by a branch instruction or a call instruction. A new branch address (W) can be stored into the program counter upon the completion of a successful branch or call instruction. If the branch instruction is not successful, then the program counter goes to the next ROM location within the current page.

In a successful call or branch execution the page address register (PA) receives its next page address from the buffer register (PB). The contents of the PB are changed by the load page instruction (LDP), which can be executed prior to the branch or call. Execution always continues on the same page unless PB is explicitly changed.

When the branch is executed successfully and when the processor is not in the call mode (CL = 0), the page buffer register is loaded into the page address register. If the contents of the page buffer register have been modified prior to the branch instruction, then this instruction is called a *long branch instruction*, since it may branch anywhere in the ROM (a long branch, BL, directive in the source program generates two instructions—LDP, load page buffer, and BR, branch). In the call mode (CL = 1), only "short" branches are possible, those staying within a given page.

Note that the normal state of the status logic is ONE. Several instructions can alter this state to a ZERO; however, the ZERO

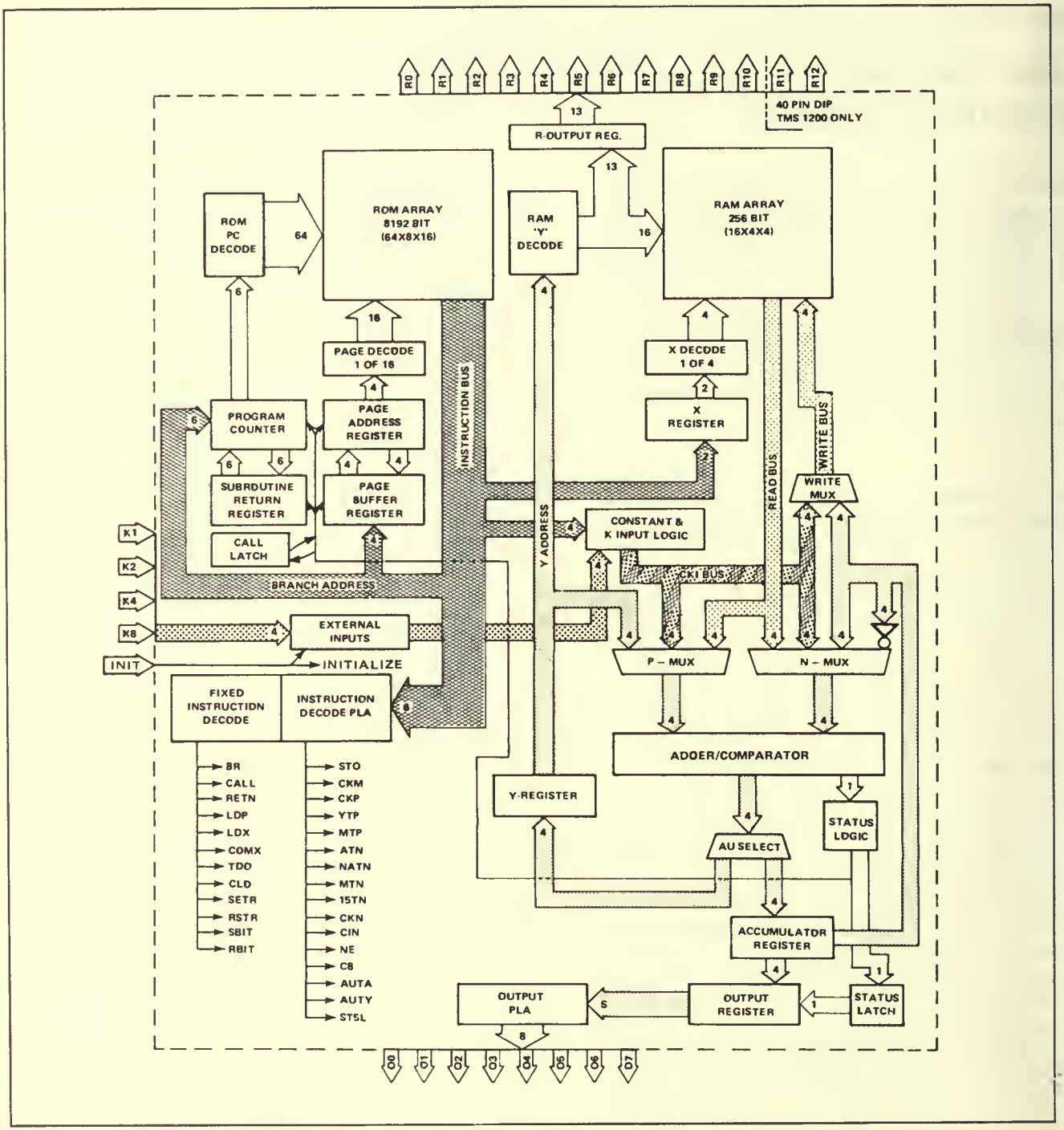


Fig. 1. TMS1000/1200 block diagram.

state lasts for only one subsequent instruction cycle (which could be during a branch or call); then the status logic will normally revert back to its ONE state (unless the following instruction resets it to ZERO).

Like branch instructions, call instructions are conditional. One level of subroutine is permitted, and a call within a call does not execute properly. In the case of a successful call when status logic equals ONE:

1 The call latch (CL) is set to ONE.

2 The contents of the page buffer register (PB) and the page address (PA) register are exchanged simultaneously.

3 The return address is stored in SR and PB. The SR address is one address ahead of the program counter when the call instruction is executed. The return address is saved for a future return instruction.

4 The branch address field of the instruction word writes into the program counter.

When a return instruction occurs:

1 The subroutine return register (containing the call instruction address plus one) is always transferred to the program counter.

2 The contents of the page buffer register (containing the page at call) is always transferred to the page address register.

3 The call mode is reset (CL = 0).

If a call instruction is executed within a previous call (no return has occurred and the call latch is still a ONE and status is a ONE), there is no transfer of the page buffer register to the page address register; instead contents of the page address register transfer to the page buffer register, although the branch address (W) loads into the program counter.

Thus a call within a call to another page will cause the return page to change, losing the correct return page address.

An X and Y address selects one four-bit RAM character, M(X,Y), this address being the storage location in the RAM matrix. The X-register can be set to a constant equal to 0 through 3 (LDX instruction), or X can be complemented (COMX instruction) to flip the address of X to the $\overline{X}$ file (e.g., 00 to 11, or 01 to 10).

The Y-register has three purposes.

1 The Y-register addresses the RAM in conjunction with the X-register for RAM character select.

2 The Y-register is a working register. The Y-register may be set to any constant between 0 and 15 (by the TCY instruction), loaded from memory (TMY instruction), loaded from the accumulator (TAY instruction), decremented (DYN), and incremented (IYC). Note that in the functional block diagram (Fig. 1), the Y-register has no inverted adder input. Thus, the Y-register cannot be subtracted from the accumulator or memory.

3 The Y-register addresses the R-output register for setting and resetting individual latches. Whenever a particular R-output needs to be set, the constant bus inputs the R's address (0 through 12) to Y (TCY instruction), and then a set R-output (SETR) instruction is executed.

The TMS1000 has two outputs:

- R-outputs used for control
- O-outputs used to transmit data

The purpose of the R-outputs is to control the following:

- External devices
- Display scans
- Input encoding
- Dedicated status logic outputs (such as overflow)

Each R-output has a latch that stores a ONE or ZERO, and each latch may be set (ONE) or reset (ZERO) individually by the set R (SETR) or reset R (RSTR) instruction. The Y-register points to which R-output is set by these instructions.

The R-output can be strobed by the ROM program to scan a key matrix (K-input). Figure 2 represents the maximum key matrix possible without external logic. A simple short from an R line to a K-input can be detected by the ROM program and interpreted as any function or data entry. Expanding the matrix is possible by external logic such as using a 4-line to 16-line decoder.

The status latch and the accumulator data are loaded into the O-output register (bottom right of Fig. 1) by a fixed instruction from the ROM (TDO) when the programmer decides to change output data. A separate instruction clears the O-output register. This instruction (CLO) causes all five output register bits to be reset to ZERO. The five bits from the O register are converted to a parallel eight-bit code by the O PLA.

The accumulator is a four-bit register that interacts with the adder, the RAM, and the output registers. The accumulator is the main working register for addition and subtraction. It is the only register which is inverted before its contents are sent to the adder for subtraction. Subtraction is accomplished by two's complement arithmetic. It is a storage register for inputs from the constant and K-input logic as well as the Y-register.

Variable data from the K-inputs is also stored via the accumula-

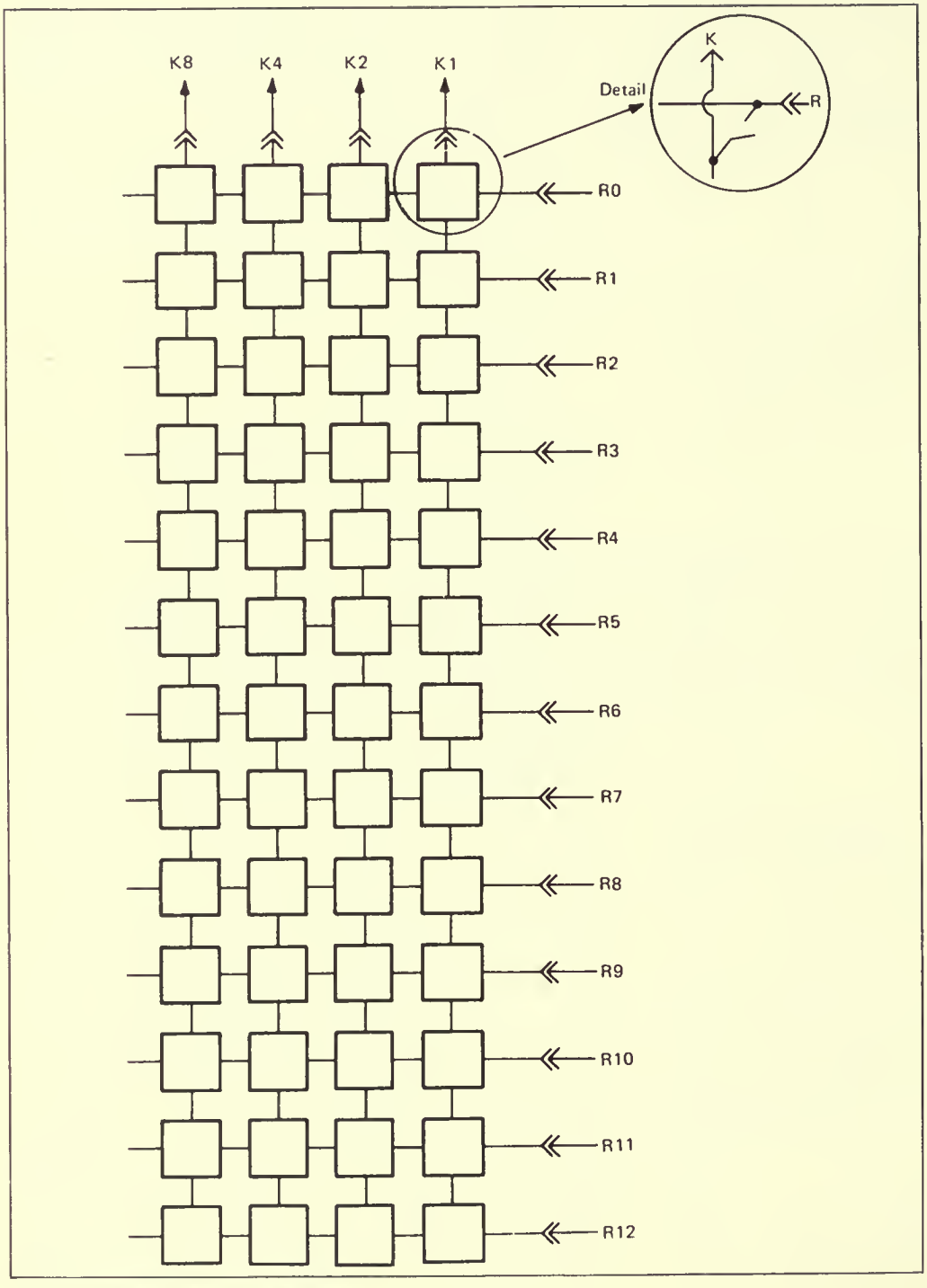


Fig. 2. Keyboard matrix connections.

tor into the RAM array. Therefore, any variable data input from the K-inputs or from the adder output must pass through the accumulator to the RAM array for storage. Likewise, any data to the O-outputs must come through the accumulator. Four accumulator register bits may be latched by the O-output register (where the status latch information is also latched) for decode by the O-output decoder.

There are 18 instructions that affect status logic, either setting it (to ONE) or resetting it (to ZERO). In turn, the status logic will permit the successful execution of a branch or call instruction (if status logic = ONE) or prevent successful execution of these instructions (if reset to ZERO). Status logic will remain at a ZERO level only for the following instruction cycle and then automatically be set to the normal ONE state (unless reset to ZERO by the next instruction).

There are two microinstructions (NE and C8) that are used by instructions affecting status. If the microinstruction C8 is used and a carry occurs in the addition of two four-bit words, the carry goes from the MSB sum to status, setting status logic to a ONE. If no carry occurs, status logic is ZERO. In a logic compare instruction (using microinstruction NE), status logic is set to ONE if the four-bit words at the N and P adder/comparator inputs are not equal; conversely, status logic is ZERO if the inputs are equal.

The status latch buffers the status-logic bit to the O-output register for decode by the O-output PLA. Status-logic output is selectively loaded into the status latch by special microinstruction STSL (used in a logical-compare test instruction that causes the status logic to output a ONE or ZERO). For example, if the test instruction YNEA (in the standard instruction set) causes status to be a ONE (if Y-register is not equal to A), then the ONE writes into the status latch. If a ZERO is output by that instruction from status logic, then the ZERO writes into the status latch.

The status latch transfers to the O-register with the accumulator bits when TDO, transfer data out, is executed.

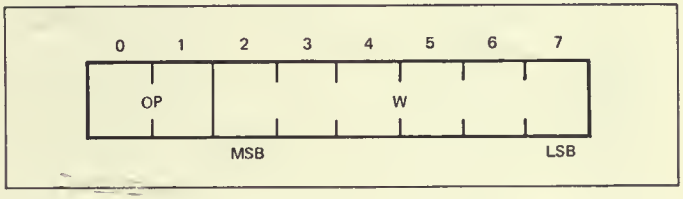
Instruction Set

Table I summarizes the standard instruction set, composed of the 12 "fixed" instructions and the 31 standard microprogrammed instructions. These standard instructions are available as a default to the user if he does not choose to redefine them by specifying a different PLA pattern.

Instruction Formats

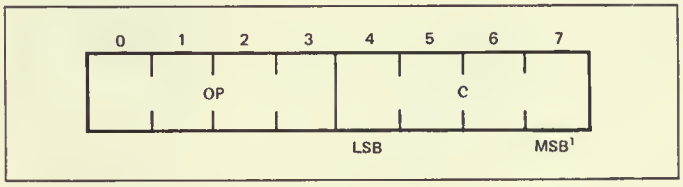
The machine instructions have been divided into four instruction formats. A format subdivides the eight bits of each instruction into fields. These fields contain the operation code and operands.

Instruction Format I:



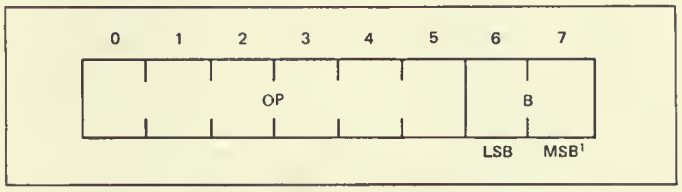
This format has a two-bit operation-code field, and the operand is a six-bit ROM-word address field. This format is used for program control by branch and call instructions. The operand, the branch address, has a value of 0 to 63.

Instruction Format II:



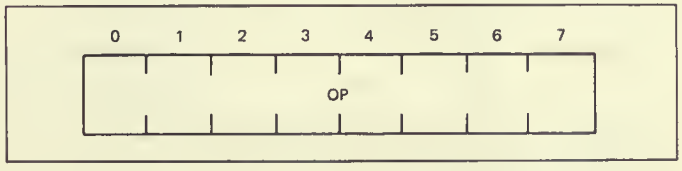
This format has a four-bit operation-code field; the operand is a four-bit constant field. This format is used for instructions that contain an immediate value that loads RAM memory or a register with a constant.

Instruction Format III:



This format has a six-bit operation code, and the operand is a two-bit RAM bit address field. This format is used for addressing a bit in a RAM word. Also, B describes the two-bit X-address operand for the LDX command.

Instruction Format IV:



[1]The constant values are reversed in this field. The assembler converts values into proper machine code format.

Table 1 TMS1000/1200 Standard Instruction Set

Function	*Mnemonic*	*Status† effect* C8	*Status† effect* NE	*Description*	*Instruction format*
Register to register	TAY	...	...	Transfer accumulator to Y-register.	IV
	TYA	...	...	Transfer Y-register to accumulator.	IV
	CLA	...	...	Clear accumulator.	IV
Transfer register to memory	TAM	...	...	Transfer accumulator to memory.	IV
	TAMIY	...	...	Transfer accumulator to memory and increment Y-register.	IV
	TAMZA	...	...	Transfer accumulator to memory and zero accumulator.	IV
Memory to register	TMY	...	...	Transfer memory to Y-register.	IV
	TMA	...	...	Transfer memory to accumulator.	IV
	XMA	...	...	Exchange memory and accumulator.	IV
Arithmetic	AMAAC	Y	...	Add memory to accumulator, results to accumulator. If carry, one to status.	IV
	SAMAN	Y	...	Subtract accumulator from memory, results to accumulator.	IV
				If no borrow, one to status.	IV
	IMAC‡	Y	...	Increment memory and load into accumulator. If carry, one to status.	IV
	DMAN‡	Y	...	Decrement memory and load into accumulator. If no borrow, one to status.	IV
	IA	...	...	Increment accumulator, no status effect.	IV
	IYC	Y	...	Increment Y-register. If carry, one to status.	IV
	DAN	Y	...	Decrement accumulator. If no borrow, one to status.	IV
	DYN	Y	...	Decrement Y-register. If no borrow, one to status.	IV
	A8AAC	Y	...	Add 8 to accumulator, results to accumulator. If carry, one to status.	IV
	A10AAC	Y	...	Add 10 to accumulator, results to accumulator. If carry, one to status.	IV
	A6AAC	Y	...	Add 6 to accumulator, results to accumulator. If carry, one to status.	IV
	CPAIZ	Y	...	Complement accumulator and increment. If then zero, one to status.	IV
Arithmetic compare	ALEM	Y	...	If accumulator less than or equal to memory, one to status.	IV
	ALEC	Y	...	If accumulator less than or equal to a constant, one to status.	II
Logical compare	MNEZ	...	Y	If memory not equal to zero, one to status.	IV
	YNEA	...	Y	If Y-register not equal to accumulator, one to status and status latch.	IV
	YNEC	...	Y	If Y-register not equal to a constant, one to status.	II
Bits in memory	SBIT	...	...	Set memory bit.	III
	RBIT	...	...	Reset memory bit.	III
	TBIT1	...	Y	Test memory bit. If equal to one, one to status.	III
Constants	TCY	...	...	Transfer constant to Y-register.	II
	TCMIY	...	...	Transfer constant to memory and increment Y.	II
Input	KNEZ	...	Y	If K-inputs not equal to zero, one to status.	IV
	TKA	...	...	Transfer K-inputs to accumulator.	IV
Output	SETR	...	...	Set R-output addressed by Y.	IV
	RSTR	...	...	Reset R-output addressed by Y.	IV
	TDO	...	...	Transfer data from accumulator and status latch to O-outputs.	IV
	CLO	...	...	Clear O-output register.	IV
RAM X addressing	LDX	...	...	Load X with a constant.	III
	COMX	...	...	Complement X.	IV
ROM addressing	BR	...	...	Branch on status = one.	I
	CALL	...	...	Call subroutine on status = one.	I
	RETN	...	...	Return from subroutine.	IV
	LDP	...	...	Load page buffer with constant.	II

†C8 (microinstruction C8 is used) — Y (Yes) means that if there is a carry out of the MSB, status output goes to the ONE state. If no carry is generated, status output goes to the ZERO state.
NE (microinstruction NE is used) — Y (Yes) means that if the bits compared are not equal, status output goes to the ONE state. If the bits are equal, status output goes to the ZERO state.
A ZERO in status remains through the next instruction cycle only. If the next instruction is a branch or call and status is a ZERO, then the branch or call is not executed.

‡Execution of the DMAN or IMAC instruction does not change (increment or decrement) the content of the addressed memory cell.

This format defines an eight-bit operation code field only. Instructions of this format have no constant operands. The instruction always performs the same action, for example, transferring the accumulator to the Y-register.

Eighteen instructions conditionally affect the machine status logic. The mnemonics for these instructions contain a one- or two-character descriptor to indicate how status logic is affected. Each descriptor (shown in Table 2) indicates the condition where status will remain set (logic ONE). The conditional instructions, branch and call, are successful only if status is set. The mnemonic descriptor therefore indicates the conditions under which an immediately following branch or call will be performed. If the instruction results do not meet the descriptor's condition, then status is reset (logic ZERO) and any immediately following branch or call will not be performed. [Status logic in the reset (ZERO) state affects only branches or calls in the next instruction cycle before returning to the normal (logic ONE) state.]

The way in which the instruction depends upon status or sets status is defined as follows:

- Set: The instruction unconditionally forces status to ONE and is not conditional upon status.
- Carry into Status: The value of the carry from the adder is transferred to status. In the subtraction instructions, carry = $\overline{\text{borrow}}$.
- Comparison Result into Status: The logical comparison value from the ALU is transferred to status (equal: ZERO to status; unequal: ONE to status).
- Conditional on Status: The instruction's execution results are conditional upon the state of the status. After the instruction is executed, status is unconditionally equal to ONE.

Implementation

The instruction timing is fixed and each instruction requires six clock cycles to execute. Each of the 43 basic instructions is defined to enable one or more microinstructions that activate control lines during one instruction cycle. These microinstructions explain the firmware bridge between software instructions and the individual logic block capabilities. A hardwired logic decoder that cannot be modified decodes 12 "fixed" basic instruction codes into 12 fixed microinstructions for output instructions, branching, subroutines, RAM X-addressing, reset and set bit instructions. The remaining 31 basic instructions activate a combination of 16 programmable microinstructions that are encoded by the instruction PLA. The concept of fixed and programmable microinstructions is used as a tool for understanding the software on the machine level and is used to increase the power of the instruction set to fit more applications (microprogramming the instruction set).

Table 2 Descriptor Action

Descriptor		*Cause/result that transfers ONE to status*
Last character in mnemonic	C	Carry out during addition or increment instructions
	N	No borrow during subtraction or decrement instructions
	Z	Zero result from 2's complement
	1	Tested memory bit is a logic ONE
Middle of mnemonic	-LE-	Is less than or equal to
	-NE-	Is not equal to

The purpose of the CKI logic (Fig. 1) is to select either the K-inputs or the four-bit constants from ROM (the C field of the instruction word) or a bit mask to go out to the CKI data bus. The constant and K-input logic is used whenever microinstructions CKP, CKN, or CKM are selected by an instruction. The data going out on the CKI bus changes for predetermined instruction values, however, and this section details what the data is and the versatility of CKI microinstructions. Since the constant and K-input logic is not changeable, it is important to understand the four separate functions CKI controls before learning how CKI microinstructions are performed. Table 3 shows the binary-decoded groupings of the instruction word and the particular output enabled by the CKI logic.

1 First, for eight hexidecimal instruction values (08 to $0F_{16}$ as listed in Table 3), the K-inputs are active. That is, the constants from the ROM are shut off, and the four-bit external-input bus (center left of Fig. 1) is made available to either the adder/comparator or the RAM. The instruction decoder determines how the available data is used.

2 The second main function is to channel constant data from the instruction bus (from ROM) to the CKI bus output (instruction values 00 to 07 and 40_{16} to $7F_{16}$ as listed in Table 3). The CKI bus is available to the P adder input, the N adder input, or the write multiplexer for the RAM as shown in Fig. 1. The constant data from the ROM can be selected by 72 possible machine instruction values, although the standard instructions use only 68 of these.

3 The constant logic is disabled (output at ZERO for values 20_{16} to $2F_{16}$).

4 A bit mask is active. For example, the bit mask as used in the test bit instruction (TBIT1) determines if a bit from the RAM is a ONE by comparing it with ZERO. The bit mask has only one ZERO in the four-bit CKI output, as determined by the B field of the instruction word (see TBIT1 in Table 3). The B field is two bits and points to the selected opening (ZERO) in the mask. Thus, if the least significant bit is to be tested, then the bit mask outputs the binary word 1110 to the CKI bus output. Then the CKI bus output goes into both sides of the adder/comparator, and the word at M(X,Y) is input simultaneously (logically ORed) with the

Table 3 Constant and K-Input Logic Truth Table

Op code (binary list)									*Op code (hex)*		*Mnemonic (standard instructions)*	*C KI out*	*C KI logic and other constant operations*	*Comment*
I	*(0)*	*(1)*	*(2)*	*(3)*	*(4)*	*(5)*	*(6)*	*(7)*						
	0	0	0	0	0	0	0	0	0	0	COMX			
	0	0	0	0	0	0	0	1	0	1	A8AAC	Y		
	0	0	0	0	0	0	1	0	0	2	YNEA			
	0	0	0	0	0	0	1	1	0	3	TAM		I(7–4) →	$I_{(7)}$ = MSB
	0	0	0	0	0	1	0	0	0	4	TAMZA		CKI bus	$I_{(4)}$ = LSB
	0	0	0	0	0	1	0	1	0	5	A10AAC	Y		
	0	0	0	0	0	1	1	0	0	6	A6AAC	Y		
	0	0	0	0	0	1	1	1	0	7	DAN	Y		
	0	0	0	0	1	0	0	0	0	8	TKA	Y		
	0	0	0	0	1	0	0	1	0	9	KNEZ	Y		
	0	0	0	0	1	0	1	0	0	A	TDO			
	0	0	0	0	1	0	1	1	0	B	CLO		$K_{1,2,4,8}$ →	K_8 = MSB
	0	0	0	0	1	1	0	0	0	C	RSTR		CKI bus	
	0	0	0	0	1	1	0	1	0	D	SETR			
	0	0	0	0	1	1	1	0	0	E	IA			
	0	0	0	0	1	1	1	1	0	F	RETN			
	0	0	0	1	C				1	...	LDP		I(7–4) → PB	No effect on CKI; only affect PB
	0	0	1	0	0	0	0	0	2	0	TAMIY			
	0	0	1	0	0	0	0	1	2	1	TMA			
	0	0	1	0	0	0	1	0	2	2	TMY			
	0	0	1	0	0	0	1	1	2	3	TYA			
	0	0	1	0	0	1	0	0	2	4	TAY			
	0	0	1	0	0	1	0	1	2	5	AMAAC			
	0	0	1	0	0	1	1	0	2	6	MNEZ			
	0	0	1	0	0	1	1	1	2	7	SAMAN		0 → CKI	
	0	0	1	0	1	0	0	0	2	8	IMAC		BUS	
	0	0	1	0	1	0	0	1	2	9	ALEM			
	0	0	1	0	1	0	1	0	2	A	DMAN			
	0	0	1	0	1	0	1	1	2	B	IYC			
	0	0	1	0	1	1	0	0	2	C	DYN			
	0	0	1	0	1	1	0	1	2	D	CPAIZ			
	0	0	1	0	1	1	1	0	2	E	XMA			
	0	0	1	0	1	1	1	1	2	F	CLA			
	0	0	1	1	0	0	B		3	...	SBIT		Bit mask →	B = 0 CKI = 1110
	0	0	1	1	0	1	B		3	...	RBIT		CKI bus	1 1101
	0	0	1	1	1	0	B		3	...	TBIT 1	Y		2 1011
														3 0111
	0	0	1	1	1	1	B		3	...	LDX		I(7–6) → X	No effect on CKI.
	0	1	0	0	C				4	...	TCY	y	I(7–4) →	I(7) = MSB
	0	1	0	1	C				5	...	YNEC	Y	CKI bus	I(4) = LSB
	0	1	1	0	C				6	...	TCMIY	Y		
	0	1	1	1	C				7	...	ALEC	Y		C → CKI bus; C = 0 to 15
	1	0	W								BR			Not used
	1	1	W								CALL			

Note: I = Instruction (op code), C = Constant, W = Branch Address, Y = Yes (CKP, CKN, or CKM microinstruction is used). PB = Page Buffer Register (ROM)

CKI bus into the P side of the adder/comparator. The compare feature of the adder/comparator is activated, and then the state of the tested bit transfers directly to status logic. The bit mask also selects RAM bits to be set or reset. For the set bit (SBIT) and reset bit (RBIT) instructions, the ZERO in the bit mask field (Table 3) also acts as a pointer to one of the four bits (identified by X- and Y-register contents) in a RAM character.

There are two PLA's in the TMS1000 series:

- The O-output PLA
- The instruction decoder PLA

In a PLA, a matrix of gates first decodes a number of binary logic inputs into a set of output lines (also called "terms"). Each term can select a combination of output lines from a second matrix of gates (see Fig. 3). Both matrices are implemented by programmable-input NAND gates (Fig. 4). Since we are concerned only with the input-to-output code conversion, positive-logic AND and OR functions are used herein.

Figure 4 shows two AND matrix terms, F_0 and F_1, which are encoding two output OR matrix terms, Q_0 and Q_1. The simplified method of presenting the same circuit is shown in Fig. 5. Each

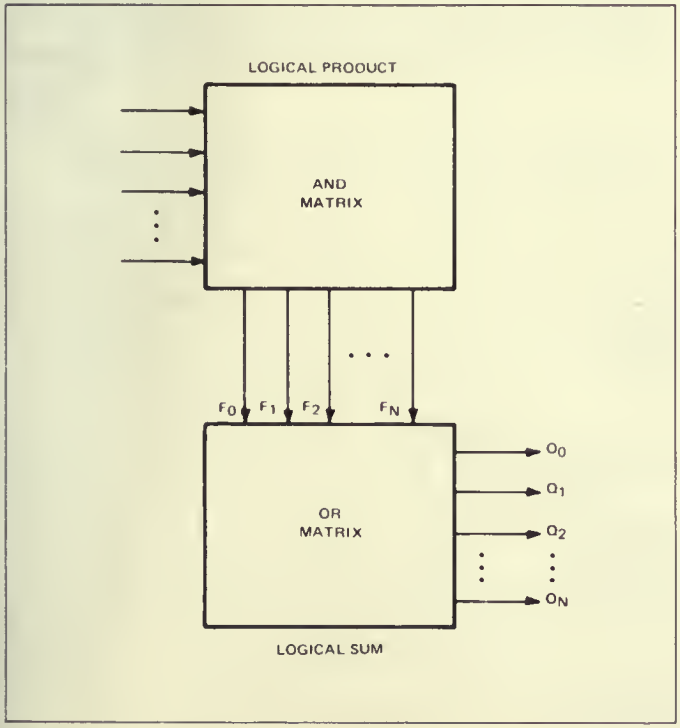


Fig. 3. PLA block diagram.

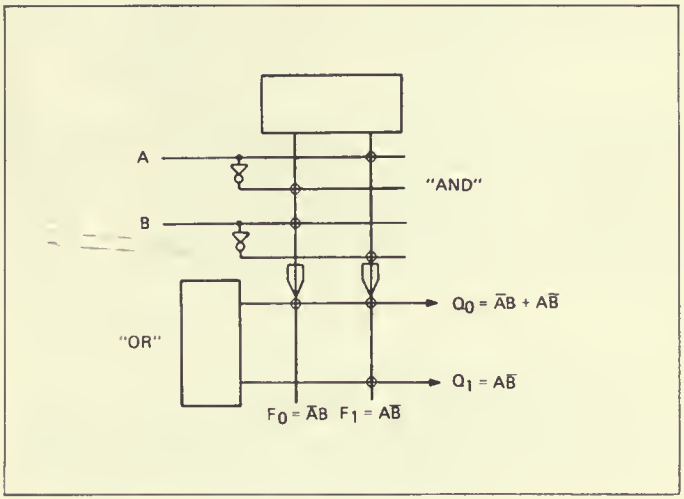


Fig. 4. Standard logic PLA circuit schematic.

circle in the diagram represents a MOSFET which selects a gate input to a matrix term.

User programming of these PLA's requires inputs to the TSM1000 simulator for O-output PLA programming and to the assembler and simulator for instruction PLA programming.

The O-output PLA determines the parallel output definition for each TMS1000 series program. Thus, a user understanding the capabilities can define an efficient output organization before designing an algorithm. The organization of the outputs is a necessary starting point for new system designs.

The O-output register sends five bits to the O-output PLA (bottom of Fig. I). Figure 6 shows the five corresponding

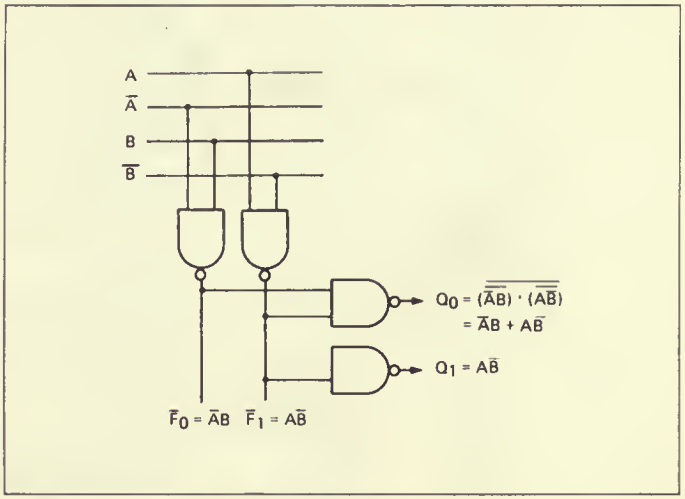


Fig. 5. Array logic equivalent schematic.

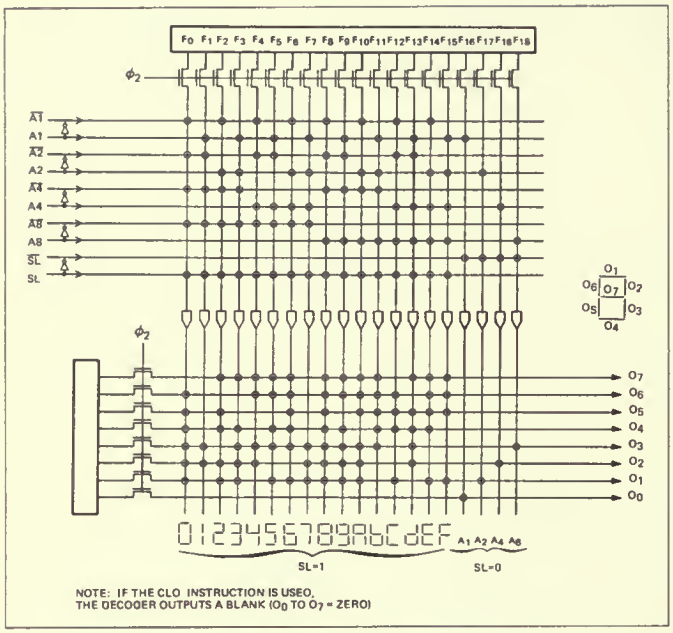


Fig. 6. Typical coding example of O-output PLA.

O-register bits (from accumulator and status latch) going into the AND matrix in true and complemented form. The AND matrix has 20 terms available for decoding a prescribed pattern of inputs to the OR matrix. The pattern is stored in the matrix by placing MOS transistors (gates) to select inputs and not placing a gate where an input is not desired.

Each AND matrix term may decode a subset of the following Boolean equation:

$$F_N = (A1 \cdot \overline{A1}) \cdot (A2 \cdot \overline{A2}) \cdot (A4 \cdot \overline{A4}) \cdot (A8 \cdot \overline{A8}) \cdot (SL \cdot \overline{SL})$$

Either the true or the complement (not both) or neither (don't care) of the two inputs enclosed in parentheses can be selected. The AND matrix may decode up to 20 of these Boolean equations.

Each OR matrix line determines the O-output pattern for each AND term used. If an AND term is true, the output selection (represented by a circle) is a subset of the following expression:

$$O \text{ output} = O_0 + O_1 + O_2 + O_3 + O_4 + O_5 + O_6 + O_7$$

If any two or more AND term equations are satisfied, then their ORed output functions are logically ORed together.

The example coding shown in Fig. 6 shows an output classified into seven-segment information and binary information. If the status latch bit is ZERO, then the PLA sends binary information out. If the status latch bit is ONE, then the PLA encodes seven-segment display information. Note that there are 20 input terms to the OR matrix; four terms encode the binary value of the accumulator bits, 16 terms encode the characters 0 to F.

The TDO instruction latches the status latch and the accumula-

Table 4 TMS1000 Series Programmable MicroInstructions

Execution sequence	*Mnemonic*	*Logic affected*	*Function*
1	CKP	P-MUX	CKI to P-adder input.
	YTP	P-MUX	Y-Reg to P-adder input.
	MTP	P-MUX	Memory (X, Y) to P-adder input.
1	ATN	N-MUX	Accumulator to N-adder input.
	NATN	N-MUX	Accumulator to N-adder input.
	MTN	N-MUX	Memory (X, Y) to N-adder input.
	15TN	N-MUX	F_{16} to N-adder input.
	CKN	N-MUX	CKI to N-adder input.
1	CIN	Adder	One is added to sum of P plus N inputs (P+N+1).
	NE	Adder/status	Adder compares P and N inputs. If they are identical, status is set to zero.
	C8	Adder/status	Carry is sent to status (MSB only).
2	STO	Write MUX	Accumulator data to memory.
	CKM	Write MUX	CKI to memory.
3	AUTA	AU select	Adder result stored into accumulator.
	AUTY	AU select	Adder result stored into Y-Reg.
	STSL	Status latch	Status is stored into status latch.

tor bits in the O register. In the case of term zero (F_0), a ONE from the status latch and ZERO from the accumulator encode the seven-segment character for zero.

Two logic blocks decode the eight-bit instructions into the various microinstructions:

- Fixed instruction decoder
- Programmable instruction PLA

The fixed instruction decoder cannot be modified and enables 12 fixed controls affecting ROM addressing, RAM X-register, output control, set bit and reset bit instructions. Every program must use these instructions with their corresponding fixed microinstructions.

The remaining 31 basic instructions in the standard set (43 basic instructions—we fixed basic instructions equal to 31 programmable instructions) have their operations determined by combining one or more microinstructions as determined by the instruction PLA.

The programmable instructions are defined to the assembler and simulator programs by default definition when the standard instructions are used. When one or more instructions are redefined, the user specifies the entire set of instruction mnemonics to the assembler, and the new PLA implementation is defined to the simulator.

Table 4 defines the operation of the programmable microinstructions, and the logic block controlled by each. In one instruction cycle the sequence of microinstruction execution is as follows:

1 Read RAM, select the inputs to the adder/comparator.
  Microinstructions: CIN, MTP, MTN, CKP, CKN, YTP, ATN, 15TN, NATAN, C8, NE

2 Write accumulator contents or CKI bus information into the RAM.
  Microinstructions: CKM, STO

3 Add or compare, then store results into the Y-register, accumulator, status logic, or status latch.
  Microinstructions: AUTY, AUTA, STSL

Thus the MTP (RAM memory contents to P-adder input) microinstruction is executed before STO (store accumulator data into RAM). The adder can perform one operation per instruction cycle. If two input buses are selected for the same side of the adder, the inputs are logically ORed together.

The programmable microinstructions are an aid to learning how instructions work. For example, the IA instruction (increment accumulator) enables three microinstructions, ATN, CIN, and AUTA:

1 ATN transfers the accumulator data to the N-adder input (P = 0).

2 CIN causes 1 to be added to the P- and N-adder inputs.

3 AUTA causes the result of the addition to be stored in the accumulator.

Knowing the hardware and how Texas Instruments combined the microinstructions explains all 31 programmable instructions. For example, the YNEC instruction activates three microinstructions.

1 CKN causes the constant from ROM (immediate operand) to go into the N-input.

2 YTP enables Y to the P-input.

3 NE sends the comparison to status.

Therefore, if Y is logically compared to a constant operand and is *not equal* to the CKI data, status equals ONE.

Figure 7 illustrates the PLA implementation designed by Texas

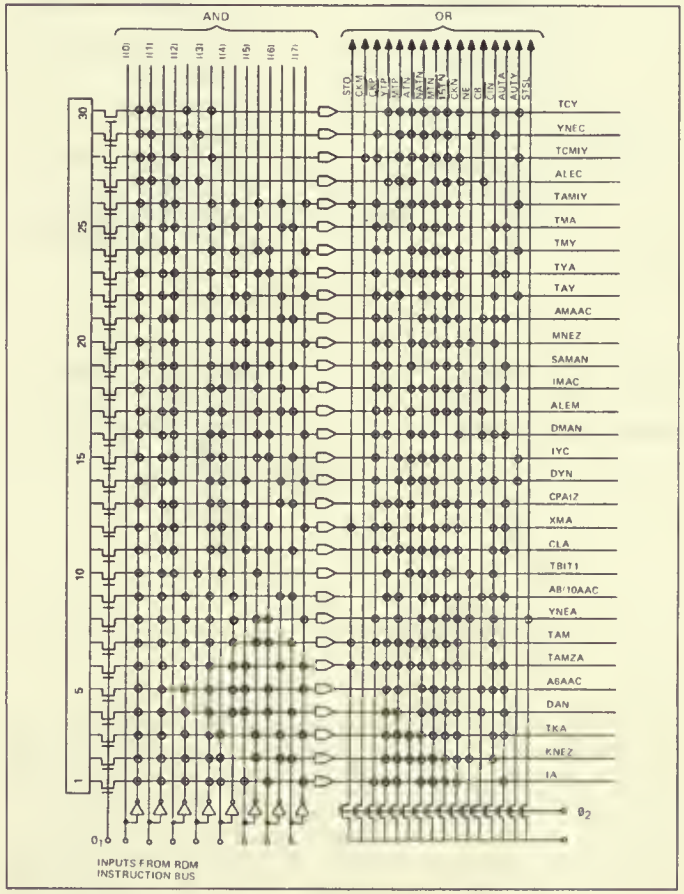


Fig. 7. TMS1000/1200 standard instruction decode PLA.

Instruments for the standard instruction set. The 31 instructions are translated by 30 PLA terms into a combination of the 16 microinstructions possible (the A8AAC and the A10AAC are combined on a single PLA line).

The instruction PLA can be reprogrammed in cases where timing or other requirements dictate an instruction redefinition. Microprogramming this PLA should be considered only when the standard definition is insufficient to accomplish the program objectives.

Addition Instruction

The following example illustrates the addition arithmetic instructions. This example shows adding a word to a BCD draft in memory. BCD correction is performed to keep the digit in the range 0 to 9. Upon exit from this routine the accumulator contains a ONE if a carry has resulted or a ZERO if no carry has resulted.

```
Label     Op code   Operand    Comment
          AMMAC                ADD CURRENT DIGIT
                               TO A
          BR        FIXUP      BRANCH IF CARRY
                               (SUM > 15)
          TAM                  TRANSFER A TO
                               MEMORY
          A6AAC                ADD 6, TEST FOR DIGIT
                               10 TO 15
          BR        CORRECT    BRANCH IF CARRY
          CLA                  CLEAR ACCUMULATOR
CONTU                          EXIT
          .
          .
          .
FIXUP     A6AAC                ADD 6 TO CORRECT
                               TO BCD
CORRECT   TAMZA                TRANSFER A TO
                               MEMORY, CLEAR A
          IA                   INCREMENT
                               ACCUMULATOR
          BR        CONTU      EXIT
          .
          .
          .
```

Input

The following example illustrates the input instructions. This example handles input from a keyboard. The keys must be sampled one row at a time. The particular row selected is determined by which R-output line is set on. This example shows sampling on row five only, and determines which of four keys on row five are depressed. If all four K-inputs are zero, no key is currently depressed. For simplicity no key-debounce logic has been included.

```
Label     Op code   Operand    Comment
          TCY       5          SET ROW 5
          SETR                 ENABLE ROW 5
          KNEZ                 TEST K INPUTS FOR NON-ZERO
          BR        INPUT      YES, GO TO INPUT
*
*NO DATA PRESENT ON INPUT LINES
*
          RSTR                 DISABLE ROW 5
          BR        CONTU      EXIT
*
*NOW STORE THE DATA FROM THE K LINES.
*
INPUT     TKA                  INPUT K LINES TO A
          RSTR                 DISABLE ROW 5
*
*
*NOW FIND WHICH KEY ON ROW 5.
*
          ALEC      1          KEY 1?
          BR        ONK1       YES
          ALEC      2          KEY 2?
          BR        ONK2       YES
          ALEC      4          KEY 4?
          BR        ONK4       YES
          BR        ONK8       MUST BE ON K8.
```

TMS1000 Display Scan

The TMS1000 is a *digit-scan* calculator chip. The displayed information is turned on one digit at a time. The segment lines for each digit are connected in parallel. The correct segment lines for a particular digit are turned on by the TMS1000 O lines and then the correct digit line (R line) is turned on to enable the illumination of that single digit. This process is continued for each digit to complete one display scan cycle, and then the entire cycle is repeated. The display is scanned as rapidly as possible to avoid flicker problems or display "breakup" when the calculator is moved. This rate is typically in the range of 150 to 300 Hz for the TMS1000.

Output

The following example illustrates the various output instructions. Four data words from memory, M(0,3), go to the O-output register. The R-outputs are used to signal which word is presented. The O-register is cleared after each word has been presented. The example assumes that a previous YNEA instruction set the status latch to ZERO.

```
Label   Op code   Operand   Comment
        .
        .
        TCY       3         SET INDEX AND COUNTER
LOOP    SETR                SET R(Y) OUTPUT STROBE
        TMA                 LOAD DIGIT INTO A
        TDO                 LOAD OUTPUT FROM A AND SL
        RSTR                RESET R(Y) OUTPUT STROBE
        CLO                 CLEAR O OUTPUT REGISTER
        DYN                 DECREMENT Y REGISTER
        BR        LOOP      LOOP UNIT Y BORROWS
        .
        .
```

Program Control

The following example illustrates the usage of the program control instructions BR, CALL, RETN and LDP.

This example illustrates using a control loop that calls a subroutine to perform a specific function. The control loop continues to call the subroutine until certain conditions are met; then control is passed to another portion of the main program in a different ROM page. This particular example calls a "shift left" routine to shift a five-word string left one word address at a time. The shift routine is called until a non-zero word is found in position M(0,3). Because the subroutine is in another page, a long call is performed by setting a new page address in the page buffer (PB) before the call.

```
Label   Op code   Operand   Comment
        LDX       0         SET RAM ADDRESS
LOOP    TCY       3          to M(0,3)
        MNEZ                M(0,3) ≠ 0;
        BR        DONE      BRANCH IF NOT EQUAL,
                            DONE
*
*SET UP TO CALL SHIFT LEFT ROUTINE
*
        LDP       5         SLRTN IS IN PAGE 5
        CALL      SLRTN     CALL SLRTN
        BR        LOOP      RETURN HERE, BRANCH
                            TO LOOP
*
DONE    LDP       4         GO TO PAGE 4
        BR        MORE      PERFORM LONG BRANCH
*
*COMMON SUBROUTINE, SLRTN, SHIFT LEFT.
*
SLRTN   TCY       0         CLEAR Y INDEX
        CLA                 CLEAR A
SWITCH  XMA                 EXCHANGE MEMORY &
                            ACCUMULATOR
        IYC                 INCREMENT Y INDEX
        YNEC      4         Y = 4? (END OF STRING)
        BR        SWITCH    CONTINUE IF NOT EQUAL
        RETN                RETURN TO CALL
```

APPENDIX 1 ISP of the TMS1000

```
TMS1200 :=
    begin

! Texas Instruments TMS 1000 Series MOS/LSI one chip microcomputer.

! References: TMS 1000 Software User's Guide
!             TMS 1000 Programmer's Reference Manual
!             The Engineering Staff of Texas Instruments Incorporated,
!             Semiconductor Group,
!             Texas Instruments Incorporated
!             P.O. Box 1443,
!             Houston, Texas 77001

! Note that the "INIT" line must be set to "1" before starting a
! simulation of the ISP.  This "feature" is a result of the
! II "INIT" implementation.

! The Output PLA and the Instruction PLA may be redefined for
! simulation.  The internal initializations should be overlayed
! by files read into the simulator after completion of
! "init.out.pla".  I.E. set ADBEAK init.out.pla before starting
! the simulation.  At the break: BEAD yourdefinition.SIM.

**MP.State**

    ROM[0:1023]<0:7>,                       ! ROM for instruction storage.
    BAM[0:63]<0:3>,                         ! BAM
        ram.bit[0:255]<> := BAM[0:63]<0:3>  ! BAM bit map

**PC.State**

        PA<0:3>,                            ! Page address register
        PB<0:3>,                            ! Page buffer register
        PC<0:5>,                            ! Program counter
        SR<0:5>,                            ! Subroutine return address
        CL<>,                               ! Call latch
        R[0:10]<>,                          ! R output register
        X<0:1>,                             !
        Y<0:3>,                             ! Pointer/storage register
        S<>,                                ! Logic status
        SL<>,                               ! Conditional branch status
        A<0:3>,                             ! Accumulator
        O<0:4>,                             ! Output buffer
        CK1.BUS<0:3>

**External.State**

        INIT<>,                             ! Init line
        X<0:3>                              ! External inputs

**Implementation.Declarations**

        N.MUX<0:3>,                         ! Multiplexer to adder
        P.MUX<0:3>,                         ! Multiplexer to adder
        ADDER<0:4>,                         ! The adder/ALU
        temp<0:3>,                          ! Temporary register
        s.trace<>,                          ! Status trace
        rom.address<0:9>,                   ! Instruction ROM address reg.
        OUT.PLA[0:31]<0:7>,                 ! Simulation of output pla

        INSTR.PLA[0:255]<0:15>,             ! Simulation of instruction PLA.
        b.rev<0:1>,                         ! Reverse bit b field.
        c.rev<0:3>                          ! Reverse bit c field.


**Instruction.Format**

    I.BUS\Instruction.Bus<0:7>,         ! Doubles as instruction register

                                        ! Format I instructions
        op.I<0:1>    := I.BUS<0:1>,     ! Opcode
        w<0:5>       := I.BUS<2:7>,     ! New branch address

                                        ! Format II instructions
        op.II<0:3>   := I.BUS<0:3>,     ! Opcode
        c<0:3>       := I.BUS<4:7>,     ! Constant (note bit reversal)

                                        ! Format III instructions
        op.III<0:5> := I.BUS<0:5>,      ! Opcode field
        b<0:1>       := I.BUS<6:7>,     !

                                        ! Format IV instructions

        op.IV<0:7>   := I.BUS<0:7>,     ! Opcode

                                        ! Format V (1000/1300 only)
        op.V<0:4>    := I.BUS<0:4>,     ! Opcode
        f<0:2>       := I.BUS<5:7>      ! Data for LOX

**PLA.Initialization**[us]

    ! The output Programmable Logic Array (PLA) translates the
    ! contents of the O register into a user defined code on the
    ! O-output lines.  The PLA initialization defined below
    ! provides encoding for driving seven segment LED displays
    ! with the characters:
    !   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, b, C, d, E, F.

    ! Additionally, BCH output can be selected under program control.

    init.out.pla :=
        begin
        OUT.PLA[0]  = '00000000;    OUT.PLA[16] = '01111110;
        OUT.PLA[1]  = '00000001;    OUT.PLA[17] = '00110000;
        OUT.PLA[2]  = '00000010;    OUT.PLA[18] = '01101101;
        OUT.PLA[3]  = '00000011;    OUT.PLA[19] = '01111001;
        OUT.PLA[4]  = '00000100;    OUT.PLA[20] = '00110011;
        OUT.PLA[5]  = '00000101;    OUT.PLA[21] = '01011011;
        OUT.PLA[6]  = '00000110;    OUT.PLA[22] = '01011111;
        OUT.PLA[7]  = '00000111;    OUT.PLA[23] = '01110000;
        OUT.PLA[8]  = '00001000;    OUT.PLA[24] = '01111111;
        OUT.PLA[9]  = '00001001;    OUT.PLA[25] = '01111011;
        OUT.PLA[10] = '00001010;    OUT.PLA[26] = '01110111;
        OUT.PLA[11] = '00001011;    OUT.PLA[27] = '00011111;
        OUT.PLA[12] = '00001100;    OUT.PLA[28] = '01001110;
        OUT.PLA[13] = '00001101;    OUT.PLA[29] = '00111101;
        OUT.PLA[14] = '00001110;    OUT.PLA[30] = '01001111;
        OUT.PLA[15] = '00001111;    OUT.PLA[31] = '01000111
        end,
    ! The Instruction PLA defined below encodes the standard II 1200
    ! (micro)programmable instruction set.  The encoding was derived
    ! from figure 2-17.2 on page 2-27 of the Programmer's Reference
    ! Manual.

    init.instr.pla :=
        begin
        INSTR.PLA["01] = '0001101111011100;             ! ABAAC
        INSTR.PLA["02] = '0010101111101001;             ! YNEA
        INSTR.PLA["03] = '1011111111001000;             ! TAM
        INSTR.PLA["04] = '1011111111001100;             ! TAMZA
        INSTR.PLA["05] = '0001101111011100;             ! A10AAC
        INSTR.PLA["06] = '0001101111011100;             ! ABAAC
        INSTR.PLA["07] = '0001101111010100;             ! DAN
        INSTR.PLA["08] = '0001111111001100;             ! IKA
        INSTR.PLA["09] = '0001111111101000;             ! KNEZ
        INSTR.PLA["0E] = '0011101111000100;             ! IA
        INSTR.PLA["20] = '1010111111000010;             ! TAMIY
        INSTR.PLA["21] = '0011011111001100;             ! IMA
        INSTR.PLA["22] = '0011011111001010;             ! IMY
        INSTR.PLA["23] = '0010111111001100;             ! IYA
        INSTR.PLA["24] = '0011101110111010;             ! IAY
        INSTR.PLA["25] = '0011001111011100;             ! AMAAC
        INSTR.PLA["26] = '0011011111101000;             ! MNEZ
        INSTR.PLA["27] = '0011010111010100;             ! SAMAN
        INSTR.PLA["28] = '0011011111010100;             ! IMAC
        INSTR.PLA["29] = '0011010111010000;             ! ALEM
        INSTR.PLA["2A] = '0011011101011100;             ! DMAN
        INSTR.PLA["2B] = '0010111111010010;             ! IYC
        INSTR.PLA["2C] = '0010111101011010;             ! DYN
        INSTR.PLA["2D] = '0011110111010100;             ! CPIAZ
        INSTR.PLA["2E] = '1011011111001100;             ! XMA
        INSTR.PLA["2F] = '0011111111001100;             ! CLA
        INSTR.PLA["3B] = '0001011110101000;             ! TBIT1
        INSTR.PLA["39] = '0001011110101000;             ! TBIT1
        INSTR.PLA["3A] = '0001011110101000;             ! TBIT1
        INSTR.PLA["3B] = '0001011110101000;             ! TBIT1
        temp = 0 next
    init.loop :=
        begin
        INSTR.PLA["40 + temp] = '0001111111001010;      ! TCY
        INSTR.PLA["50 + temp] = '0010111110101000;      ! YNEC
        INSTR.PLA["60 + temp] = '0110111111000010;      ! TCMIY
        INSTR.PLA["70 + temp] = '0001110111010000 next  ! ALEC
        temp = temp + 1 next
        IF temp NEQ 0 => RESTART init.loop
        end
        end

**Service.Routines**{us}

    ! Access routine to translate the O register through the Output PLA.

    activate.out.pla(O<0:4>)<0:7> := (activate.out.pla = OUT.PLA[O]),

    ! Access routine to translate instructions through the instruction PLA.

    activate.instr.pla(I.BUS<0:7>)<0:15> :=
                                (activate.instr.pla = INSTR.PLA[I.BUS])

**Instruction.Interpretation**[us]

    start[main] :=                              ! Main control loop
        begin
        IF init =>                              ! Initialization sequence
            begin
            init.instr.pla(); init.out.pla();
            PC = O = A = CL = 0; PA = PB = '1111;
            S = 1; !NIT = 0
            end next
        s.trace = 0;
        rom.address =
            PA@(PC<2:5>@(PC<1> eqv PC<5>)@(PC<0> xor PC<1> eqv PC<5>)) next
        I.BUS = ROM[rom.address] next
        b.rev = b<1>@b<0>;
        c.rev = c<3>@c<2>@c<1>@c<0>;
        PC = PC<1:5> @ ((PC<0> eqv PC<1>) xor (PC<1:5> eql '11111)) next
```

APPENDIX 1 (cont'd.)

```
        DECODE I.BUS =>                          ! CKI.BUS determination
            begin
            "00:"07 := CKI.BUS = c.rev,          ! Load from instruction.
            "08:"0F := CKI.BUS = K,              ! Input from external lines.
            "20:"2F := CKI.BUS = 0,              ! Zero.
            "30:"3B := CKI.BUS = ('1110 SLR b.rev),
            "4B:"7F := CKI.BUS = c.rev,          ! Load from instruction.
          otherwise := no.op()
            end next
        DECODE I.BUS =>                          ! Fixed instruction decode
            begin
            "B0 := COMX(),                       ! Complement X
            "BA := TDO(),                        ! Transfer: O = A
            "00 := CLO(),                        ! Clear O-output
            "0C := RSTR(),                       ! Reset R[Y]
            "B0 := SETR(),                       ! Set   R[Y]
            "BF := RETN(),                       ! Subroutine Return
        "10:"1F := LDP(),                        ! Load Page Buffer (constant)
        "30:"33 := SBIT(),                       ! Set memory bit
        "34:"37 := RBIT(),                       ! Reset memory bit
        "3C:"3F := LDX(),                        ! Load X (constant)
        "80:"BF := BR(),                         ! Branch on status = 1
        "C0:"FF := CALL(),                       ! Call subroutine (status = 1)
      otherwise := microexecution()
            end next
        IF s.trace => RESTART start next
        S = 1 next
        RESTART start
    end
**Instruction.Execution**{us}

    BR :=                                        ! Branch on status = 1
        begin
        DECODE S =>
            begin
            0 := S = 1,
            1 := (IF CL eql '0 => PA = PB; PC = w)
            end
        end,

    CALL :=                                      ! Call subroutine,
        begin                                    ! conditional on status
        DECODE S&CL =>
            begin
            '10 := (SR = PC; temp = PA next
                    PA = PB next
                    PB = temp; PC = w; CL = 1),
            '11 := (PC = w; PB = PA),
      otherwise := S = 1
            end
        end,

    RETN :=                                      ! Return from subroutine
        begin
        IF CL => PC = SR next
        PA = PB; CL = 0
        end,

    LDP  := (PB = c.rev),                        ! Load page buffer
    LDX  := (X = b.rev),                         ! Load x with constant
    COMX := (X = not X),                         ! Complement x
    TDO  := (activate.out.pla(SL&A)),            ! Transfer to output
    CLO  := (activate.out.pla(0)),               ! Clear output register
    SETR := (IF Y leq 10 => R[Y] = 1),           ! Set R[Y] to 1
    RSTR := (IF Y leq 10 => R[Y] = 0),           ! Set R[Y] to 0
    SBIT := (ram.bit[X@Y@b.rev] = 1),            ! Set memory bit
    RBIT := (ram.bit[x&y&b.rev] = 0)             ! Reset memory bit

**Microinstruction.Execution**{us}

    microexecution :=
        begin
        activate.instr.pla(I.BUS); P.MUX = 0; N.MUX = 0 next
        IF      activate.instr.pla<0>  => STO  := RAM[X@Y] = A;
        IF      activate.instr.pla<1>  => CKM  := RAM[X@Y] = CKI.BUS next
        IF not activate.instr.pla<2>  => CKP  := P.MUX = CKI.BUS;
        IF not activate.instr.pla<3>  => YTP  := P.MUX = Y;
        IF not activate.instr.pla<4>  => MTP  := P.MUX = RAM[X@Y];
        IF not activate.instr.pla<5>  => ATN  := N.MUX = A;
        IF not activate.instr.pla<8>  => NATN := N.MUX = not A;
        IF not activate.instr.pla<7>  => MTN  := N.MUX = RAM[X@Y];
        IF not activate.instr.pla<8>  => IN15 := N.MUX = "F;
        IF not activate.instr.pla<9>  => CKN  := N.MUX = CKI.BUS next
        ADDER = P.MUX + N.MUX next
        IF      activate.instr.pla<10> => NE   := (S = (P.MUX neq N.MUX);
                                                   s.trace = 1);
        IF not activate.instr.pla<12> => CIN  := ADDER = P.MUX + N.MUX + 1 next
        IF      activate.instr.pla<11> => C8   := (S = ADDER<0>;
                                                   s.trace = 1) next
        IF      activate.instr.pla<13> => ADTA := A = ADDER<1:4>;
        IF      activate.instr.pla<14> => ADTY := Y = ADDER<1:4>;
        IF      activate.instr.pla<15> => STSL := SL = S
        end
    end          ! End of TMS1000
```