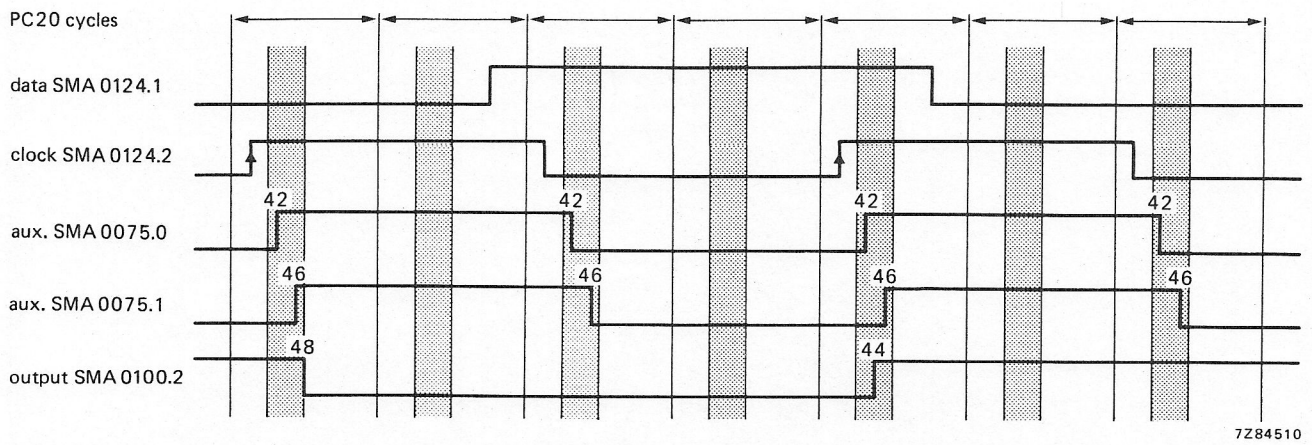


Fig. 3-11 Waveforms explaining operation of program (shaded areas) in Example 3-3. Numbers refer to lines of program where variable changes. Output 0100.2 assumed to be HIGH where waveforms start.

Different auxiliary addresses, 0075.0 and 0075.1, are used for the TRIG logic instructions in the two instruction blocks; if not, a positive edge of the clock pulse could only be detected on line 42, not on line 46, and condition '1' required for executing the SET 0 instruction would never occur. The flip-flop can be made to respond to a negative-going clock edge by using the ANDNT logic instruction on lines 41 and 45.

A shorter software D flip-flop program is obtained by using the JFRF instruction; see Example 3-16 in Section 3.2.13.

3.2.5 The FETCH BIT, FETCH DIGIT and FETCH CONSTANT execute instructions

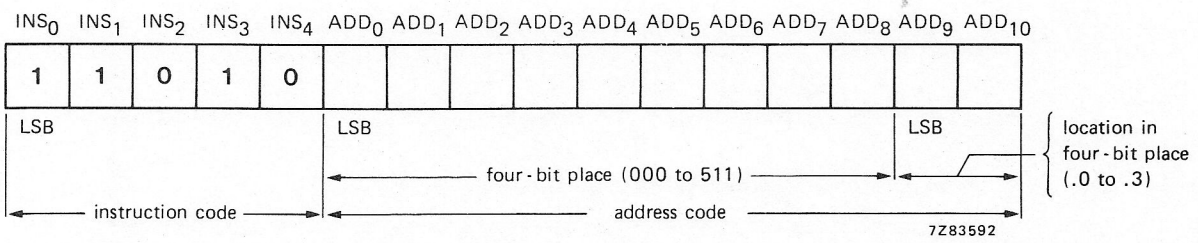
denomination	inst. no.	instruction code					mnemonic	condition for fulfilment	1-bit or 4-bit inst.
		INS_0 (LSB)	INS_1	INS_2	INS_3	INS_4 (MSB)			
FETCH BIT	11	1	1	0	1	0	FTCHB	1	1-bit
FETCH DIGIT	13	1	0	1	1	0	FTCHD	1	4-bit
FETCH CONSTANT	12	0	0	1	1	0	FTCHC	1	4-bit

The FETCH BIT, FETCH DIGIT and FETCH CONSTANT execute instructions are conditional; they will only be executed when the condition is '1' (see the above table). Table 3-5 illustrates the data flow which occurs when the FETCH instructions are active. These instructions are used to load the A and B registers for further processing; the data may originate in the scratchpad memory (FETCH BIT and FETCH DIGIT execute instructions) or the program memory (FETCH CONSTANT execute instruction).

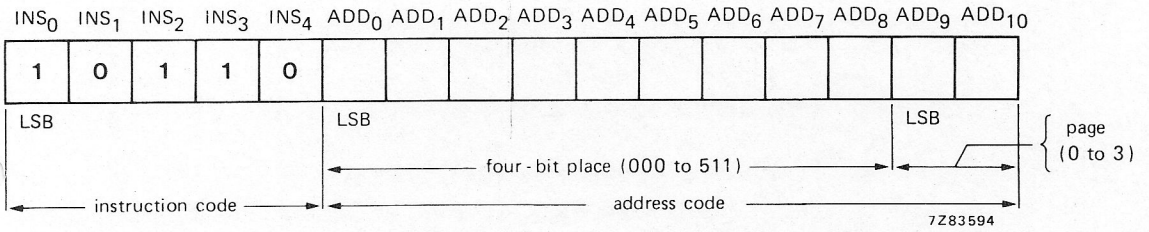
Since the A and B registers have a sixteen-bit capacity, they will act as a sixteen-bit memory for the single-bit FETCH BIT execute instruction and as a four-by-four-bit memory for the four-bit FETCH DIGIT and FETCH CONSTANT execute instructions.

TABLE 3-5 Data flow for FETCH instructions.

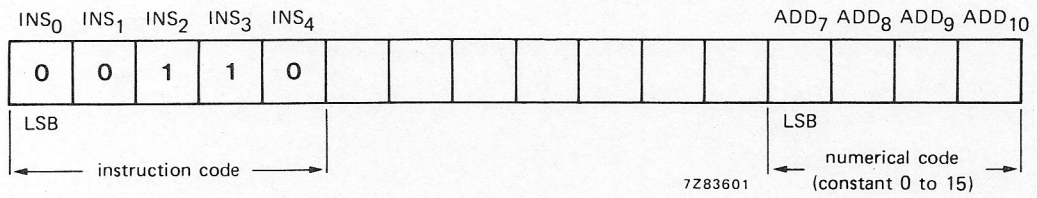
instruction	data flow	note
FTCHB, FTCHD	SM → A & B reg.	first FETCH inst. in series clears A and B registers
FTCHC	PM → A & B reg.	



FTCHB *Instruction no. 11. This instruction causes the data of the one-bit scratchpad place addressed in the program word* (see the above figure) *to be transferred to the A and B registers.* Sixteen successive FTCHB instructions will fill these registers, the first FTCHB instruction in the series clearing the remaining fifteen one-bit places in the A and B registers. If less than sixteen FTCHB instructions are given, the remaining places in the A and B registers will be left in the reset state. A seventeenth FTCHB instruction added in the series would overrule the first FTCHB instruction.



FTCHD *Instruction no. 13. This instruction causes the data of the four-bit scratchpad place addressed in the program word* (see the above figure) *to be transferred to the A and B registers.* Four successive FTCHD instructions will fill these registers, the first FTCHD instruction in the series clearing the remaining three four-bit places in the A and B registers. If less than four FTCHD instructions are given, the remaining places in the A and B registers will be left in the reset state. A fifth FTCHD instruction added in the series would overrule the first FTCHD instruction.



FTCHC *Instruction no. 12. This instruction causes the four-bit data (number 0 to 15) specified in the program word* (see the above figure) *to be transferred to the A and B registers.* Four successive FTCHC instructions will fill these registers, the first FTCHC instruction in the series clearing the remaining three four-bit places in the A and B registers. If less than four FTCHC instructions are given, the remaining places in the A and B registers will be left in the reset state. A fifth FTCHC instruction added in the series would overrule the first FTCHC instruction.

Examples using the FETCH instructions will be given in the following sections.

3.2.6 The STORE BIT and STORE DIGIT execute instructions

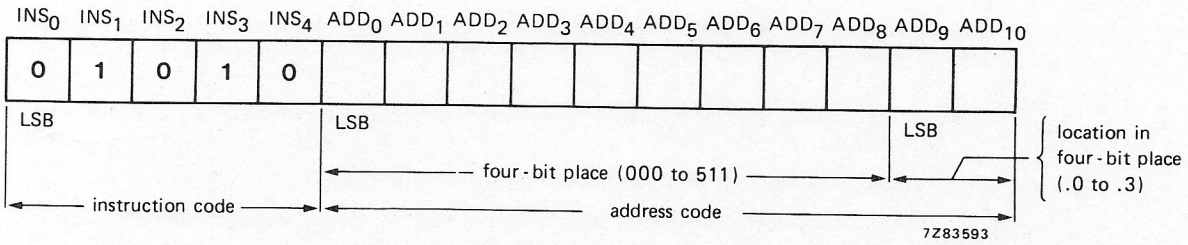
denomination	inst. no.	instruction code					mnemonic	condition for fulfilment	1-bit or 4-bit inst.
		INS_0 (LSB)	INS_1	INS_2	INS_3	INS_4 (MSB)			
STORE BIT	10	0	1	0	1	0	STRB	1	1-bit
STORE DIGIT	14	0	1	1	1	0	STRD	1	4-bit

The STORE BIT and STORE DIGIT execute instructions are conditional; they will only be active when the condition is '1', as shown in the above table. Both instructions cause the data to be transferred to the scratchpad memory. These instructions cannot stand alone but must be preceded by one or more execute instructions identifying the data source; this is shown in Table 3-6.

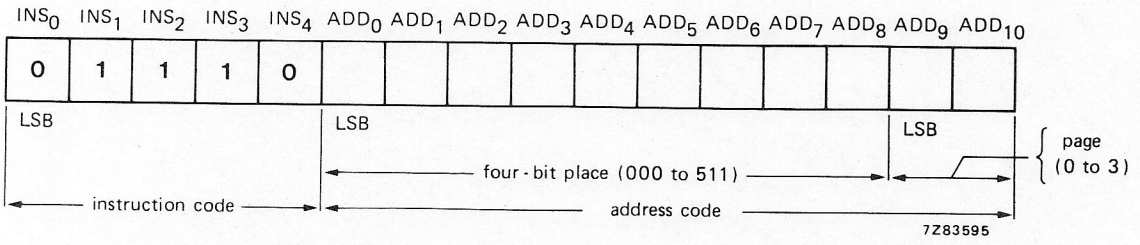
TABLE 3-6 Data flow for STORE instructions.

preceding inst.	STORE inst.	data flow	notes	section
FTCHB, FTCHD, FTCHC	STRB, STRD	A reg. → SM	data in A & B reg. retained.	3.2.5
ADD, SUBTR, MULT	STRB, STRD	A reg. → SM	arithmetic result in A reg. retained.	3.2.7
DIV	STRB, STRD	M/Q reg. → SM	quotient in M/Q reg. retained; remainder stored in A register.	3.2.7
COMP	STRB, STRD	comp. reg. → SM	comp. reg. set to "=" condition.	3.2.8
CNTD, CNTU	STRB	state reg.* → SM		3.2.9
SHFTL, SHFTR	STRB	overflow reg.** → SM		3.2.10

* In BCD counter. ** In shift register.



STRB *Instruction no. 10. This instruction causes one-bit data in the work register* (Table 3-6) *to be transferred to the scratchpad place described in the program word* (see the above figure). The contents of the A or M/Q register will be transferred when sixteen successive STRB instructions are given. A seventeenth STRB instruction added in the series would cause the first place in the A or M/Q register to be read out again.



STRD *Instruction no. 14. This instruction causes four-bit data in the work register* (Table 3-6) *to be transferred to the scratchpad place described in the program word* (see the above figure). The contents of the A or M/Q register will be transferred when four successive STRD instructions are given. A fifth STRD instruction added in the series would cause the first place in the A or M/Q register to be read out again.

The following should be observed:

- Where FETCH instructions are followed by STORE instructions, the data that have been entered *first* into the A register will be transferred *first* to the scratchpad memory *(First-In-First-Out register).*
- The STORE instruction following the arithmetic execute instructions ADD, SUBTR, MULT or DIV causes the arithmetic operation to be carried out. If a series of STRD instructions is given, the *first* instruction will cause the *least significant digit* to be extracted, then the other digits will follow in the order of increasing significance. In the case of a series of STRB instructions the *least significant bit in the least significant digit* will *first* be transferred to the scratchpad followed by bits of increasing significance, similarly to the case of the STRD instructions.

Examples 3-4 to 3-6 illustrate the use of the FETCH and STORE instructions.

Example 3-4 Collecting scratchpad data.

In the following program part FTCHB instructions are followed by a STRD instruction. This program can be used to store data dispersed in the scratchpad on output addresses.

```
line   prog. word
 :       :
0343  AND     0016.0          When input 0016.0 is HIGH,
0344  FTCHB   0100.3  (a)     (0100.3) → A & B reg.,
0345  FTCHB   0148.2  (b)     (0148.2) → A & B reg.,
0346  FTCHB   0135.2  (c)     (0135.2) → A & B reg.,
0347  FTCHB   0135.0  (d)     (0135.0) → A & B reg.,
0348  STRD    0200    (abcd)  (A reg.) → 0200.
 :       :
```

The FTCHB and STRD instructions are conditional so they must be preceded by one or more logic instructions; here, the logic instruction "AND 0016.0" is used. On lines 344 to 347 the contents, (a), (b), (c) and (d) respectively, of the addressed one-bit places are stored in the A and B registers and, on line 348, transferred from the A register to the new address 0200. The contents of the scratchpad source locations and of the A and B registers are retained. Figure 3-12 shows what happens. It is seen that the bit fetched *first* (line 344) is stored at location .0 of SMA 0200. Further, the following bits (lines 345, 346 and 347) are transferred to SMAs 0200.1, 0200.2 and 0200.3, respectively.

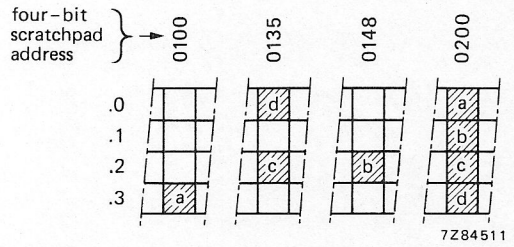


Fig. 3-12 Example 3-4: Collecting scratchpad data. Data bit a on address 0100.3 is transferred to address 0200.0 denoted a; data bit b on address 0148.2 is transferred to address 0200.1 denoted b. etc.

Example 3-5 Dispersing scratchpad data.

Here, the FTCHD instruction is followed by STRB instructions. This program is useful to distribute data in the scratchpad.

```
line   prog. word
 :       :
0501  AND     0004.3          When input 0004.3 is HIGH,
0502  FTCHD   0017    (pqrs)  (0017) → A & B reg.,
0503  STRB    0083.1  (p)  ┐
0504  STRB    0127.0  (q)  ├  (A reg.) →
0505  STRB    0083.3  (r)  ├  0083.1, 0127.0, 0083.3 & 0113.3
0506  STRB    0113.3  (s)  ┘
 :       :
```

On line 502 the contents of SMA 0017 are stored in the A and B registers and transferred on lines 503 to 506 from the A register to the allocated scratchpad places. Here, the contents of the source location, SMA 0017, and that of the A and B registers will also stay. Figure 3-13 illustrates that the bit on location .0 of SMA 0017 will be stored *first* (line 503) followed by bits on locations 0017.1, 0017.2 and 0017.3 (STRB instructions on lines 504 to 506).

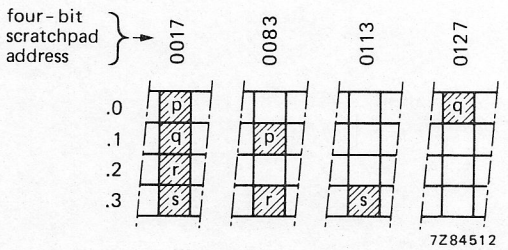


Fig. 3-13 Example 3-5: Dispersing scratchpad data. Data bit p on address 0017.0 is transferred to address 0083.1 marked p; data bit q on address 0017.1 is transferred to address 0127.0 marked q, etc.

Example 3-6 Resetting scratchpad places.

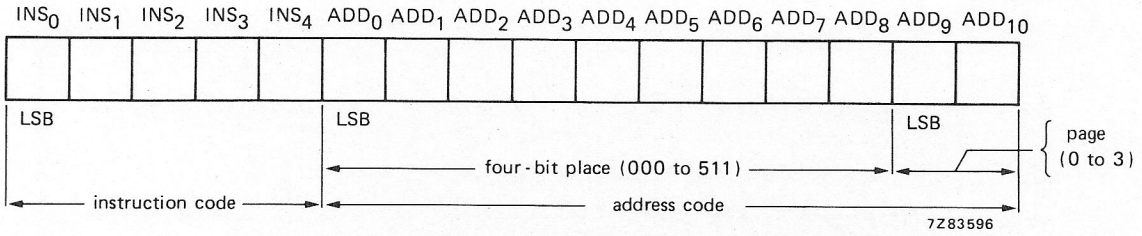
Scratchpad places can be reset using the execute instruction "FTCHC 0" followed by a string of STRD and/or STRB instructions. In the program here the STRD instruction is used to reset four-bit scratchpad places.

```
line   prog. word
 :       :
0600  AND     0010.1    When input 0010.1 is HIGH,
0601  FTCHC   0         clear A and B reg. and store 0000 on 1st addr.
0602  STRD    0300      (1st "address" in A reg.) = 0000 → 0300,
0603  STRD    0301      (2nd "address" in A reg.) = 0000 → 0301,
0604  STRD    0302      (3rd "address" in A reg.) = 0000 → 0302,
0605  STRD    0304      (4th "address" in A reg.) = 0000 → 0304,
0606  STRD    0306      (1st "address" in A reg.) = 0000 → 0306,
0607  STRD    0307      (2nd "address" in A reg.) = 0000 → 0307,
      etc.
```

Clearly, the chain of STRD instructions can have an arbitrary length. To reset single-bit scratchpad places, STRB instructions can be used in a similar way.

3.2.7 The ADD, SUBTRACT, MULTIPLY and DIVIDE execute instructions

denomination	inst. no.	instruction code					mnemonic	condition for fulfilment
		INS_0 (LSB)	INS_1	INS_2	INS_3	INS_4 (MSB)		
ADD	20	0	0	1	0	1	ADD	1
SUBTRACT	21	1	0	1	0	1	SUBTR	1
MULTIPLY	22	0	1	1	0	1	MULT	1
DIVIDE	23	1	1	1	0	1	DIV	1



The four-bit arithmetic execute instructions ADD, SUBTRACT, MULTIPLY and DIVIDE describe the type of arithmetic operation. They are conditional as they will only become effective when the condition '1' occurs (see the table). An arithmetic program takes the basic form illustrated here.

a. LOGIC instructions; the program will be executed when the condition is '1'.

b. FETCH instructions to load the A and B registers (Section 3.2.5).

c. ARITHMETIC instructions to load the proper work register.

d. STORE instructions to transfer the arithmetic result to the scratchpad (Section 3.2.6); *the first of the series of STORE instructions causes the arithmetic operation to be executed.*

The following general notes apply:

- Numbers can only be processed in BCD code – digits 0 to 9.
- Fractional and negative numbers cannot be programmed.
- To signalize arithmetic overflow, the overflow bit place, SMA 0000.0 (Table 3-3, Section 3-2), will be set to '1' level if the arithmetic result is greater than 9999 (register capacity exceeded) or negative.
- The *first* in the series of arithmetic instructions will reset the overflow bit place SMA 0000.0 to '0' level.

ADD *Instruction no. 20. This construction causes the data of the four-bit scratchpad place addressed in the program word* (see the figure at the head of this section) *to be transferred to the B register.* Four successive ADD instructions will fill the B register, the first ADD instruction in the chain clearing the remaining three four-bit places of this register.

The ADD operation is expressed as:

Augend + Addend = Sum.

Table 3-7 illustrates the flow of arithmetic data.

Notes:

1. Because the A and B registers have a four-by-four-bit memory capacity, *up to four* FETCH, ADD or STRD instructions can be programmed.
2. Since the *first* set of four data bits in transit is seen by the data processor as the *least significant digit,* the series of FETCH, ADD and STRD instructions should start with that digit, the other digits to follow in the order of increasing significance.
3. If the augend has p digits and the addend q digits, the required number of STRD instructions is p + 1 or q + 1, whichever is greater; however, not more than four STRD instructions must be given – see Note 1.
4. After the addition has been carried out, the sum will be retained in the A register for further arithmetic operations (*no* multiplication).
5. To carry out a subaddition whose result need not be stored in the scratchpad, a single STRD (or STRB) instruction can be given to make the result available in the A register; see Example 3-9.

TABLE 3-7 Data flow for addition.

instruction	data flow	notes
FTCHD, FTCHC	augend from SM or PM → A & B reg.	first FETCH inst. clears A & B reg.
ADD	addend from SM → B reg.	first ADD inst. clears B reg. and resets SMA 0000.0
STRD	sum from A reg. → SM	first STRD inst. sets SMA 0000.0 if sum > 9999.

Example 3-7 shows how an addition is programmed.

Example 3-7 Addition.

```
line   prog. word
 :       :
0673  AND     0100.2        When input 0010.2 is HIGH,
0674  FTCHD   0065   LSD ┐
0675  FTCHD   0064       ├ fetch augend,
0676  FTCHD   0063       ┘
0677  ADD     1409   LSD ┐
0678  ADD     1408       ├ fetch addend and
0679  ADD     1407       ┘
0680  STRD    1507   LSD ┐
0681  STRD    1506       │
0682  STRD    1505       ├ store sum on SMAs 1504 to 1507.
0683  STRD    1504       ┘
 :       :
```

In the program, arbitrary rather than successive scratchpad places may be chosen. Because the augend and addend can each have three digits, four STRD instructions will be needed to store the result in the scratchpad (see Note 3 above). Note that each string of execute instructions starts with the least significant digit (Note 2). Figure 3-14 shows the allocation of the scratchpad addresses.

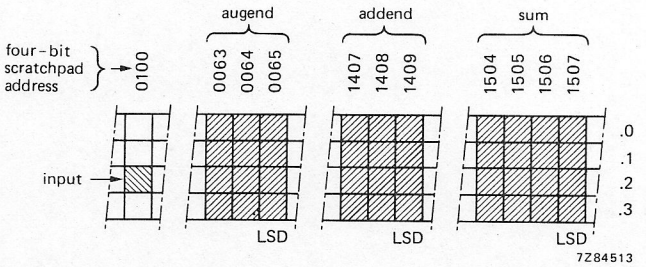


Fig. 3-14 Example 3-7: Allocation of scratchpad addresses for addition.

SUBTR *Instruction no. 21. This instruction causes the data of the four-bit scratchpad place addressed in the program word* (see the figure at the head of this section) *to be transferred to the B register.* Four successive SUBTR instructions will fill the B register, the first SUBTR instruction in the string clearing the remaining three four-bit places of this register.

The SUBTRACT operation takes the form:

Minuend – Subtrahend = Difference.

Table 3-8 shows the flow of arithmetic data.

TABLE 3-8 Data flow for subtraction.

instruction	data flow	notes
FTCHD, FTCHC	minuend from SM or PM → A & B reg.	first FETCH inst. clears A & B reg.
SUBTR	subtrahend from SM → B reg.	first SUBTR inst. clears B reg. and resets SMA 0000.0.
STRD	difference from A reg. → SM	first STRD inst. sets SMA 0000.0 if difference < 0.

Notes:

1. Because of the four-by-four-bit memory capacity of the A and B registers, *up to four* FETCH, SUBTR or STRD instructions can be given.
2. As in the case of addition, the series of FETCH, SUBTR and STRD instructions must open with the *least* significant digit, the others to follow in the order of increasing significance.
3. The required number of STRD instructions (max. four) is equal to the number of digits in the minuend.
4. After the subtraction has been carried out, the difference will be retained in the A register for further arithmetic operations (*no* multiplication).
5. To carry out a subtraction, the result of which need not be stored in the scratchpad memory, a single STRD (or STRB) instruction is given to make the result available in the A register.

Two examples are given here, one including a simple subtraction, the other combining addition and subtraction with overflow indication.

Example 3-8 Subtraction.

```
line   prog. word
 :       :
0411  AND    0012.0      ┐
0412  TRIG   0012.3      ┘ When input 0012.0 goes HIGH,
0413  FTCHD  2003  LSD   ┐
0414  FTCHD  2002        │
0415  FTCHD  2001        │ fetch minuend,
0416  FTCHD  2000        ┘
0417  SUBTR  0025  LSD   ┐ fetch subtrahend and
0418  SUBTR  0024        ┘
0419  STRD   2003  LSD   ┐
0420  STRD   2002        │ store difference on
0421  STRD   2001        │ SMAs 2000 to 2003.
0422  STRD   2000        ┘
 :       :
```

Also here, scratchpad places can be chosen at will. Since the minuend may contain four digits, four STRD instructions must be programmed (Note 3). Note that the difference is stored at the addresses of the minuend; therefore, the program must be executed once only on demand (TRIG instruction). The program is useful for stock keeping. Each time, as new data are entered on input addresses 0024 and 0025 (number of stock items removed) the subtraction will be carried out and the balance stored on output addresses 2000 to 2003. Figure 3-15 shows the allocation of the scratchpad addresses.

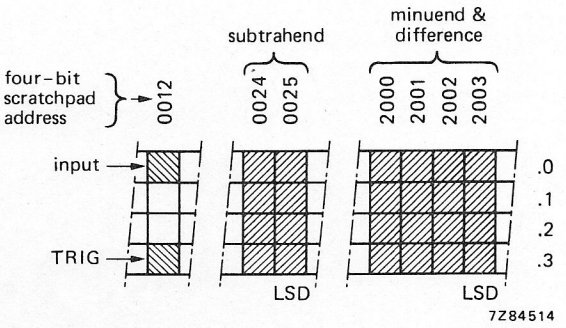


Fig. 3-15 Example 3-8: Allocation of scratchpad addresses for subtraction.

Example 3-9 Addition and subtraction with overflow detection.

```
line   prog. word
 :       :
0100  AND     0004.3   When input 0004.3 is HIGH,
0101  FTCHD   0093     LSD ┐
0102  FTCHD   0092         │
0103  FTCHD   0091         ├─fetch augend
0104  FTCHD   0090         ┘
0105  ADD     0101     LSD ┐
0106  ADD     0100         ├─fetch addend and
0107  STRD*   0113     LSD   store sum in A register.
0108  ANDNT   0000.0   If sum > 9999, that is (0000.0) = 1,
0109  EQLNT   0012.0   alarm output 0012.0 goes HIGH; when sum ≤ 9999,
0110  SUBTR   0105     LSD ┐
0111  SUBTR   0104         ├─fetch subtrahend and
0112  STRD    0113     LSD ┐
0113  STRD    0112         │
0114  STRD    0111         ├─store result on SMAs 0110 to 0113.
0115  STRD    0110         ┘
0116  AND     0000.0   If difference < 0
0117  EQL     0012.0   alarm output 0012.0 goes HIGH.
```

This program consists of three instruction blocks; it is expressed as:

Augend + Addend – Subtrahend = Result.

The STRD instruction on line 107 causes the addition to be carried out with the sum – intermediate result – stored in the A register for the subsequent subtraction and the least significant digit being transferred to SMA 0113. The least significant digit stored here will always be overruled by the result of the subtraction, which is programmed on lines 110 to 115. Subtraction will occur provided that there is no overflow in the addition (sum equal to 9999, or smaller). Note, the complement of the contents of overflow - bit place SMA 0000.0 must be programmed ("ANDNT 0000.0") as the subsequent execute instructions SUBTR and STRD are active on condition '1'. Alarm output 0012.0 will become HIGH if the sum exceeds 9999 (line 109) or if the difference becomes negative (line 117). Figure 3-16 shows the distribution of the scratchpad addresses.

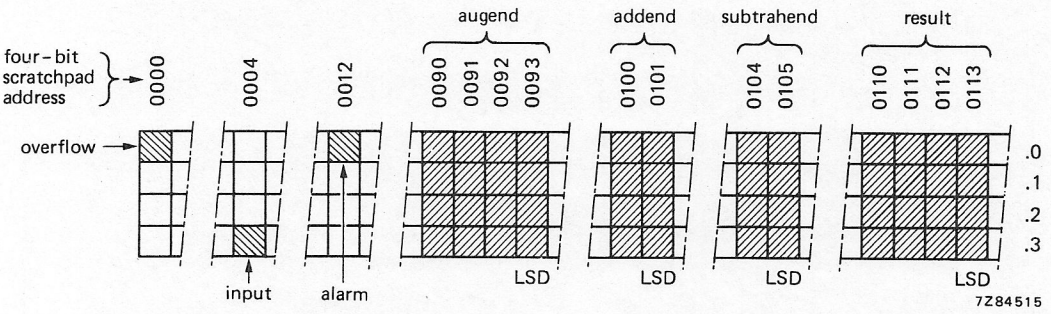


Fig. 3-16 Example 3-9: Allocation of scratchpad addresses for addition and subtraction.

* A STRB instruction has the same effect but its execution time is about 1,3 μs longer; see Table 3-12 in Section 3.3.

MULT *Instruction no. 22. This instruction causes the data of the four-bit scratchpad place addressed in the program word* (see the figure at the head of this section) *to be transferred to the M/Q register.* Four successive MULT instructions will fill the M/Q register, the first MULT instruction in the chain clearing the remaining three four-bit places of this register *as well as all four-bit places of the A register.*

The MULTIPLY operation is as follows:

Multiplicand x Multiplier = Product.

Table 3-9 shows the data flow.

TABLE 3-9 Data flow for multiplication.

instruction	data flow		notes
FTCHD, FTCHC	multiplicand from SM or PM	→ A & B reg.	first FETCH inst. clears A & B reg.
MULT	multiplier from SM	→ M/Q reg.	first MULT inst. clears M/Q reg. *and A reg.* and resets SMA 0000.0.
STRD	product from A reg.	→ SM	first STRD inst. sets SMA 0000.0 if product > 9999.

Notes:

1. Because of the four-by-four-bit memory capacity of the A, B and M/Q registers, up to four FETCH, MULT or STRD instructions can be given.
2. As in the previous cases, the series of FETCH, MULT and STRD instructions should open with the *least* significant digit, the others to follow in the order of increasing significance.
3. The required number of STRD instructions (max. four) is equal to the sum of the number of digits of multiplicand and multiplier.
4. It follows from the definition of the MULT instruction and Table 3-9 that the first MULT instruction in the series will clear the A register for multiplication and thus *destroy* the result of a previous arithmetic operation. Therefore, if it is required for multiplication, the result must be stored in the scratchpad, then fetched again for the multiplication part of the program (Example 3-10).
5. After the multiplication has been carried out, the product will stay in the A register for further arithmetic operations (*no* multiplication).
6. To carry out a submultiplication (result not necessarily stored in the scratchpad) a single STRD (or STRB) instruction is given to make the result available in the A register.

Example 3-10 Addition followed by multiplication.

The following algebraic expression is programmed here:

$(P + Q)R = T.$

```
line   prog. word
 :       :
1000  AND    0004.2   When input 0004.2 is HIGH,
1001  FTCHD  0013     LSD ┐
1002  FTCHD  0012         ┘ fetch P,
1003  ADD    0101     LSD ┐
1004  ADD    0100         ┘ fetch Q,
1005  STRD   0202     LSD ┐
1006  STRD   0201         ├ safeguard result P + Q,
1007  STRD   0200         ┘
1008  FTCHD  0202     LSD ┐
1009  FTCHD  0201         ├ fetch P + Q,
1010  FTCHD  0200         ┘
1011  MULT   0210         fetch R (A register cleared) and
1012  STRD   0223     LSD ┐
1013  STRD   0222         ├ store result (P + Q)R = T on
1014  STRD   0221         │ SMAs 0220 to 0223.
 015  STRD   0220         ┘
 :       :
```

Although the value, P + Q, of the sum is not of immediate interest here, it has to be stored on auxiliary places 0200 to 0202 (see lines 1005 to 1007) because the MULT execute instruction on line 1011 will clear the A register for multiplication. Figure 3-17 shows the allocation of the scratchpad addresses.

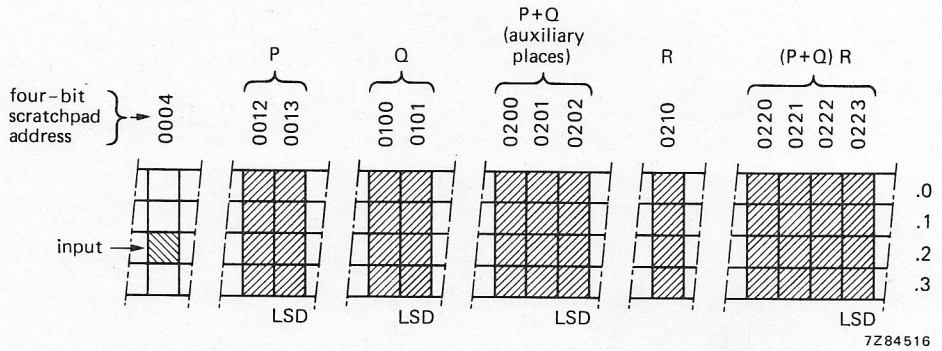


Fig. 3-17 Example 3-10: Allocation of scratchpad addresses for addition and multiplication.

DIV *Instruction no. 23. This instruction causes the data of the four-bit scratchpad place addressed in the program word* (see the figure at the head of this section) *to be transferred to the B register.* Four successive DIV instructions will fill the B register, the first DIV instruction in the series clearing the remaining three four-bit places of this register.

The DIVIDE operation is expressed as:

$$\frac{\text{Dividend}}{\text{Divisor}} = \text{Quotient} + \frac{\text{Remainder}}{\text{Divisor}}$$

Table 3-10 gives the flow of data for division.

TABLE 3-10 Data flow for division.

instruction	data flow	notes
FTCHD, FTCHC	dividend from SM or PM → A & B reg.	first FETCH inst. clears A & B reg.
DIV	divisor from SM → B reg.	first DIV inst. clears B reg. and resets SMA 0000.0.
STRD	quotient from M/Q reg. → SM	first STRD inst. sets SMA 0000.0 if divisor = 0. Remainder left in A reg.

Notes:

1. Because of the four-by-four-bit memory capacity of the A, B and M/Q registers up to four FETCH, DIV or STRD instructions are allowed.
2. As in all previously discussed arithmetic operations, the series of FETCH, DIV and STRD instructions must start with the *least* significant digit, the others to follow in the order of increasing significance.
3. The number of STRD instructions (max. four) needed to extract the quotient is equal to the number of digits of the dividend.
4. Since the remainder is retained in the A register, it can be extracted by subsequently programming an addition (adding zero); this is shown in Example 3-11.

Example 3-11 Division and extraction of remainder.

The program below follows the formula given above.

```
line   prog. word
 :        :
0090  AND     0100.3   When input 0100.3 is HIGH,
0091  FTCHC   0        ┐
0092  STRD    1011     ┘ prepare addition (0000 → SMA 1011),
0093  FTCHD   1003  LSD ┐
0094  FTCHD   1002     │
0095  FTCHD   1001     │ fetch dividend,
0096  FTCHD   1000     ┘
0097  DIV     1005  LSD ┐ fetch divisor,
0098  DIV     1004     ┘
0099  STRD    1009  LSD ┐
0100  STRD    1008     │
0101  STRD    1007     │ store quotient on SMAs 1006 to 1009 and,
0102  STRD    1006     ┘
0103  ADD     1011     ┐
0104  STRD    1011  LSD │ by adding zero, extract remainder from
0105  STRD    1010     ┘ A reg. and store on SMAs 1010 & 1011.
 :        :
```

All of the program is comprised in a single instruction block. As the remainder is retained in the A register, it can be extracted by adding zero (line 103). In the program, SMA 1011 (for storage of the LSD of the remainder – line 104) doubles as an auxiliary address for numeral zero (lines 92 and 103). Constant '0' must be fetched at the beginning of the program; if the FTCHC instruction were to occur after division, the remainder stored in the A register would be destroyed (A register cleared). Note, the required number of STRD instructions to obtain the remainder is equal to the number of digits contained in the divisor. The distribution of the scratchpad addresses is as shown in Fig. 3-18.

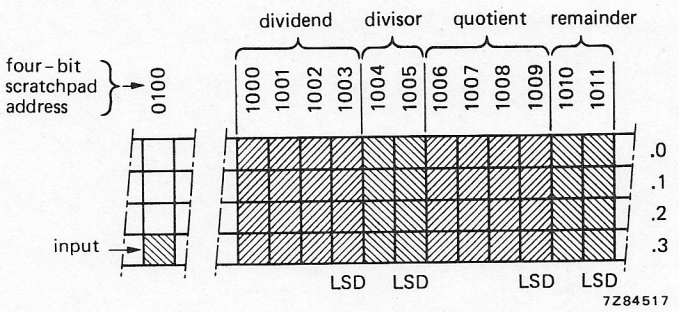
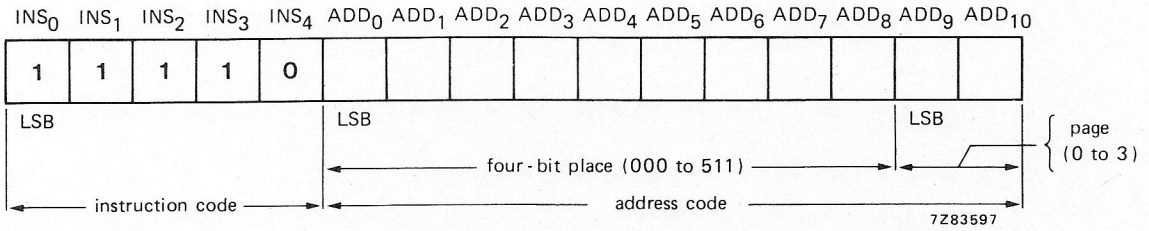


Fig. 3-18 Example 3-11: Allocation of scratchpad addresses for division.

3.2.8 The COMPARE execute instruction

denomination	inst. no.	instruction code					mnemonic	condition for fulfilment
		INS_0 (LSB)	INS_1	INS_2	INS_3	INS_4 (MSB)		
COMPARE	15	1	1	1	1	0	CMP	1



As the above table shows, the four-bit COMPARE execute instruction becomes active on condition '1'. The program for comparison contains FETCH, COMPARE and STORE instructions; Table 3-11 shows what happens when the program is executed.

TABLE 3-11 Data flow for comparison.

instruction	data flow		notes
FTCHD, FTCHC	data from SM or PM	→ A & B reg.	first FETCH inst. clears A & B reg.
COMP	data from SM	→ comparator	(A reg.) and (comparator) compared and result stored in comp. reg.
STRD, STRB	(comp. reg.)	→ SM	comp. reg. reset to '=' state.

COMP *Instruction no. 15. This instruction causes the data of the four-bit scratchpad places addressed in the program word* (see the above figure) *to be transferred to the comparator and to be compared with four bits previously loaded into the A register.* By using four COMP instructions in succession, all sixteen bits previously loaded into the A register, and the sixteen bits of the scratchpad places addressed by these instructions, will be compared. The result of comparison is stored in the four-bit compare register for use in an immediately following compare operation, or for subsequent storage in the scratchpad.

Table 3-12 shows the contents of the compare register after execution of the COMP instruction.

TABLE 3-12 State of the compare register as a result of comparison.

comparison		state of compare register			
		bit 0	bit 1	bit 2	bit 3
(A reg)**	= (SMA)*	1	0	0	0
(A reg)**	< (SMA)*	0	1	0	1
(A reg)**	> (SMA)*	0	0	1	1

* Contents of scratchpad place(s) addressed by COMP inst.

** All or part of contents of A register.

The STRD execute instruction following the COMP instruction(s) transfers the contents of the compare register to the addressed four-bit place; bit 0 to bit place .0, bit 1 to bit place.1, etc. So the following will hold (see Table 3-12):

for (A reg) = (SMA), location .0 will assume '1' level;

for (A reg) < (SMA), location .1 will assume '1' level;

for (A reg) > (SMA), location .2 will assume '1' level;

for (A reg) ≠ (SMA), location .3 will assume '1' level.

After execution of the STRD instruction the compare register will be left in the state 1 0 0 0 (Table 3-12). So the COMP instruction must be followed by *not more than one* STRD instruction.

If a STRB instruction is used instead of STRD, bit 0 in the compare register will be loaded into the one-bit scratchpad place addressed by the STRB instruction. So, from Table 3-12:

for (A reg) = (SMA), bit level '1' will be loaded;

for (A reg) ≠ (SMA), bit level '0' will be loaded.

Here too, because the compare register is reset to the equality state, *not more than one* STRB instruction must be programmed.

The *first* set of four data bits transferred by any pair of FTCHD and COMP instructions (Example 3-12) or transferred by the first in a series of FTCHD instructions and the first in the succeeding series of COMP instructions (Example 3-13) will be seen by the data processor as the *least significant* digit. So comparison must start with that digit, comparisons of the others to follow in the order of increasing significance.

Because the memory capacity of the A register is four-by-four bits, up to four successive FETCH instructions can be given. This will also apply to the COMP instructions as a number of FETCH instructions is followed by the same number of COMP instructions.

An equality following an inequality will *not* change the contents of the compare register; see Example 3-12. The STORE instruction terminating comparison will reset the compare register to the equality state to allow a new comparison to occur.

The COMP instruction is intended to compare two four-bit groups out of two bit patterns, the bit weights in each group having the values 1 (least significant bit), 2, 4 and 8 (most significant bit). Typical applications are in vehicle tagging and identification systems. Two application examples follow. In Example 3-12 two bit-patterns of equal length are compared. The program of Example 3-13 is suitable to compare a number of bit patterns of up to sixteen bits with a reference pattern having the same number of bits.

If notation is in BCD code (digits 0 to9), the bit patterns will take the form of numbers.

Example 3-12 Comparison of two bit patterns.

```
line   prog. word
  :        :
0604  AND     0100.3   When input 0100.3 is HIGH,
0605  FTCHD   0114     load LSD of pattern P into A reg. and
0606  COMP    2010     compare with LSD of pattern Q in comparator,
0607  FTCHD   0113     load next digit of pattern P into A reg. and
0608  COMP    2009     compare with next digit of pattern Q in comparator,
0609  FTCHD   0112
0610  COMP    2008
0611  FTCHD   0111
0612  COMP    2007
  :        :
0619  FTCHD   0107     load MSD of pattern P into A reg. and
0620  COMP    2003     compare with MSD of pattern Q in comparator and
0621  STRD    1003     store result on SMA 1003.
  :        :
```

Clearly, this program can be extended to compare bit patterns of arbitrary length. The operation of the program is as follows. If, for instance, on lines 607 and 608 an inequality is found, with the weight of the digit in pattern P greater than that in pattern Q, the contents of the compare register will become 0 0 1 1 (Table 3-12) as a result of comparison. If now on lines 609 and 610 an equality is met, the contents of the compare register will stay. For the weight of the digit in pattern P on line 611 found to be smaller than that of pattern Q on line 612, the contents of the compare register will change to 0 1 0 1. At the end of the program, line 621, the ultimate result will be stored on SMA 1003, bit location .0 assuming '1' level for bit patterns of equal weight, bit location .1 assuming '1' level if the weight of bit pattern P is smaller than that of bit pattern Q, etc. Figure 3-19 shows the distribution of the scratchpad addresses.

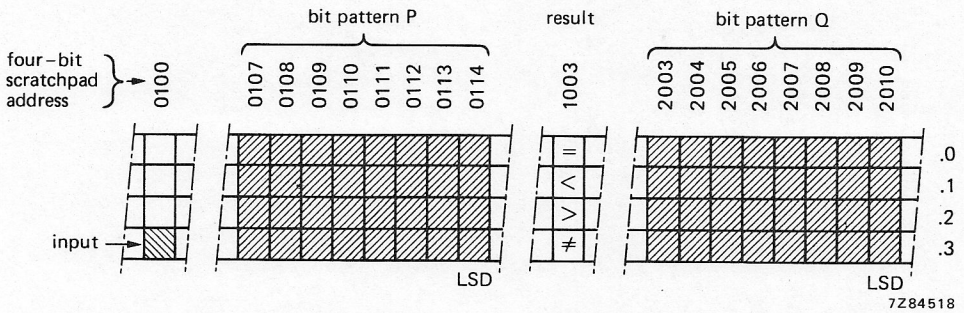


Fig. 3-19 Example 3-12: Allocation of scratchpad addresses for comparison of two 8 X 4 bit patterns. Locations 1003.0, 1003.1, 1003.2 and 1003.3 assume '1' level for P = Q, P < Q, P > Q, P ≠ Q, respectively.

Example 3-13 Comparison of bit patterns with a reference pattern.

```
line   prog. word
 :       :
0093  AND     0012.3   When input 0012.3 is HIGH,
0094  FTCHD   0064     LSD ┐
0095  FTCHD   0065         ├ load reference pattern R into A reg.,
0096  FTCHD   0066         ┘
0097  COMP    0068     LSD ┐
0098  COMP    0069         ├ compare pattern S with reference R and
0099  COMP    0070         ┘
0100  STRB    0067.0       store result on SMA 0067.0,
0101  COMP    0071     LSD ┐
0102  COMP    0072         ├ compare pattern T with reference R and
0103  COMP    0073         ┘
0104  STRB    0067.1       store result on SMA 0067.1,
0105  COMP    0074     LSD ┐
etc.                       ├ compare pattern U with reference R, etc.
                           ┘
```

Here, in a single instruction block, twelve-bit patterns are compared with the reference pattern previously loaded into the A register. Each comparison is terminated with the STRB instruction, which will store the result in the addressed one-bit scratchpad place and reset the compare register in the equality state for the next comparison. If a given pattern has a weight equal to that of the reference, '1' will be stored in the selected one-bit place; if the weights are unequal, '0' will be stored. One more FTCHD and COMP instruction can be added to each of the chains of FTCHD and COMP instructions for comparison of sixteen-bit patterns. Any number of bit patterns can be compared with the reference pattern. See Fig. 3-20 for distribution of scratchpad places.

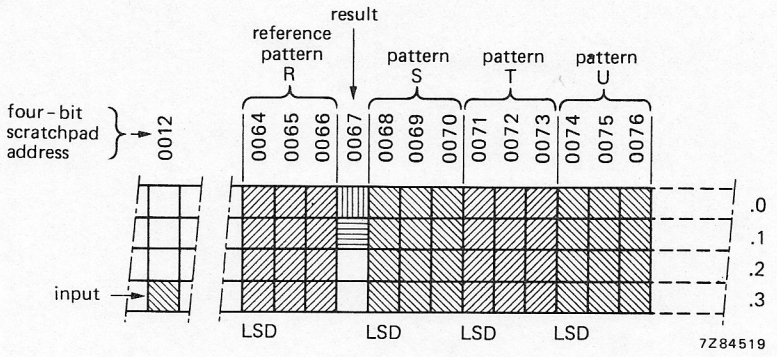
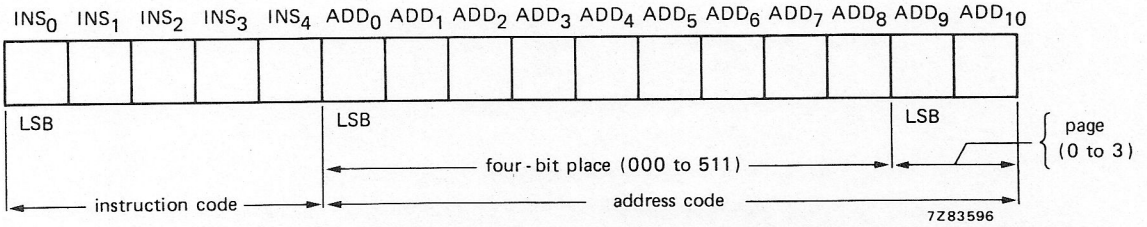


Fig. 3-20 Example 3-13: Allocation of scratchpad addresses for comparison with reference pattern.

3.2.9 The COUNT DOWN and COUNT UP execute instructions

denomination	inst. no.	instruction code					mnemonic	condition for fulfilment
		INS_0 (LSB)	INS_1	INS_2	INS_3	INS_4 (MSB)		
COUNT DOWN	6	0	1	1	0	0	CNTD	1
COUNT UP	7	1	1	1	0	0	CNTU	1



The four-bit COUNT DOWN and COUNT UP execute instructions are used to build software decade counters and timers. As the table shows, condition '1' is required for the execution of these instructions.

CNTD *Instruction no. 6. This instruction causes the data, numerical value D, of the four-bit scratchpad place specified in the program word* (see the figure above) *to be transferred to the first four-bit place (BCD counter) of the M/Q register; in this register the value of these data will be decreased by 1 for re-entry into the same scratchpad address.*

CNTU *Instruction no. 7. This instruction causes the data, numerical value D, of the four-bit scratchpad place specified in the program word* (see the figure above) *to be transferred to the first four-bit place (BCD counter) of the M/Q register; in this register, the value of these data will be increased by 1 for re-entry into the same scratchpad place.*

A software multi-decade counter is formed by programming a string of CNTD or CNTU instructions. The first instruction in the string will always be executed on condition '1'. However, in a string of CNTD instructions, any following instruction will only become effective when, in the immediately preceding instruction, the numerical value of the data has changed from 0 to 9 ("borrow"). Likewise, in the case of a string of CNTU instructions, any following instruction will not be obeyed unless, in the immediately preceding instruction, the numerical value of the data has changed from 9 to 0 ("carry").

The number of stages of a software multi-decade counter is limited by the size of the scratchpad or program memory only. UP and DOWN counters can be programmed in a single instruction block, all of them responding to the same logic instructions, which open that instruction block.

A chain of CNTD or CNTU instructions may be concluded by the STRB instruction. The STRB instruction causes the contents, '0' or '1', of the state register in the BCD counter part of the M/Q register to be stored in the one-bit place addressed by that instruction. When, as a result of the preceding CNTD instructions, all decades have assumed the value 0, the state register will be set to the '1' state; if not, it will assume the '0' state. Similarly, when as a result of the preceding CNTU instruction all decades have assumed the value 9, the state register will take the '1' state, state '0' occurring for all other values. So the state register indicates whether the counter is empty or full. Figure 3-21 illustrates the case for a two-decade DOWN counter; state bit '1' will appear for counter position 00 (counter empty). In Fig. 3-22 (two-decade UP counter) state bit '1' will occur for counter position 99 (counter filled). The state bit is used for programming timing circuits.

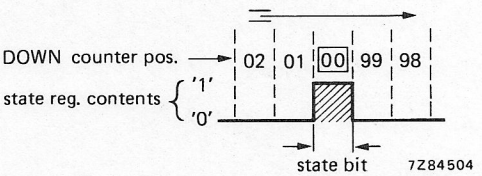


Fig. 3-21 Contents of state register for two-decade DOWN counter. State bit '1' occurs when counter is empty (position 00).

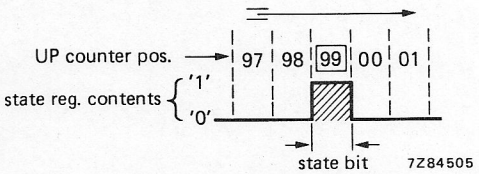


Fig. 3-22 Contents of state register for two-decade UP counter. State bit '1' occurs when counter is full (position 99).

Example 3-14 is a program of a two-decade software counter.

Example 3-14 Two-decade DOWN counter.

```
line   prog. word
 :        :
0763  AND    0001.0   When 0,1 s clock pulse occurs,
0764  TRIG   0012.1 ┐
0765  CNTD   0027   ┘ make one count (units);
0766  CNTD   0028     when borrow occurs make one count (tens);
0767  STRB   0012.0   store contents of state register on SMA 0012.0.
 :        :
```

The TRIG instruction is necessary to prevent racing. Note that the contents of the state register written in SMA 0012.0 will become '1' for counter position 00 and '0' for all other positions. Figure 3-23 shows the counter waveforms. State bit '1' will appear when the counter has arrived in position 00 (counter empty); this feature is used in timing circuits.

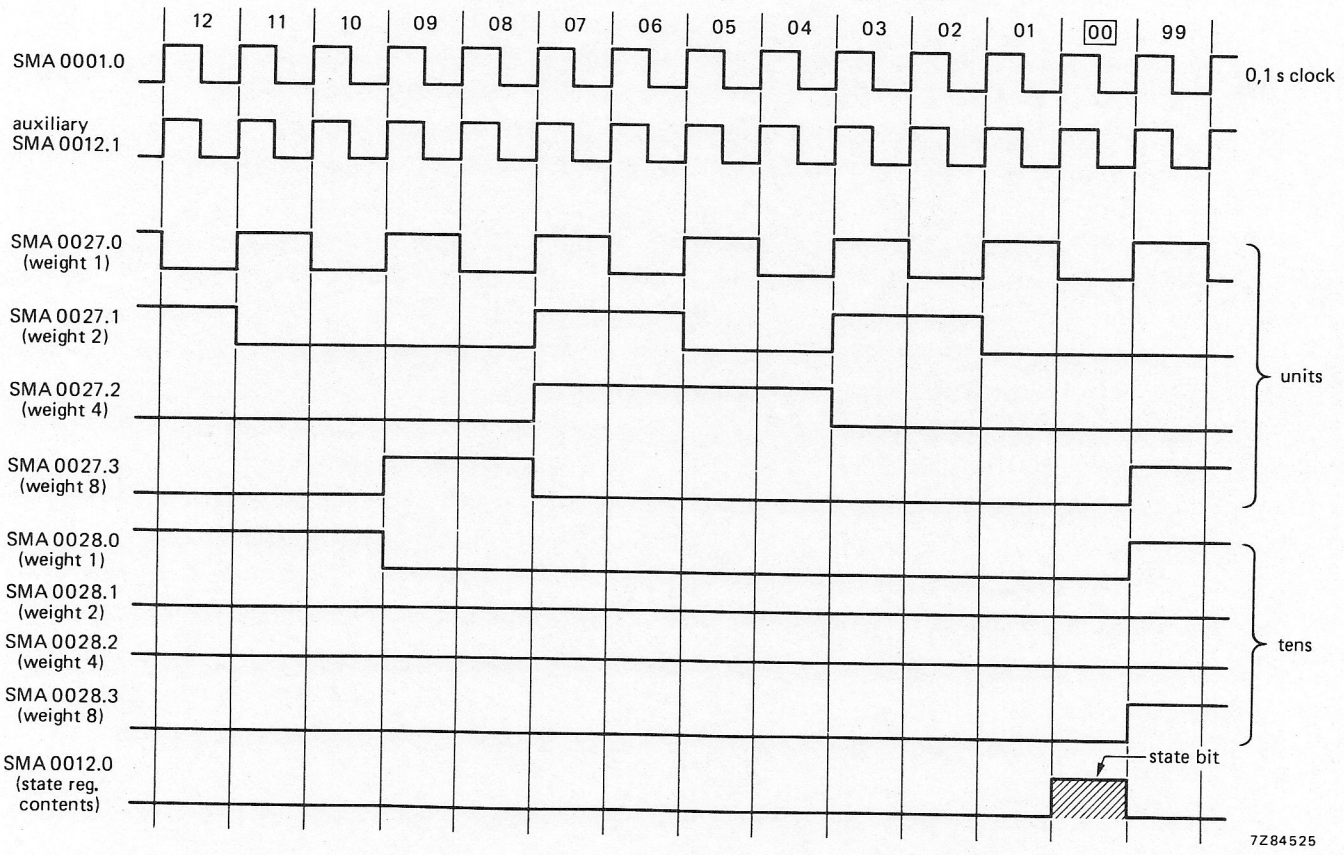
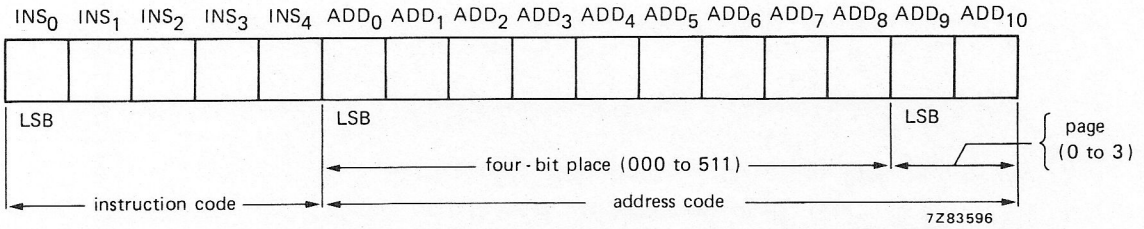


Fig. 3-23 Example 3-14: Waveforms of two decade DOWN counter.

3.2.10 The SHIFT LEFT and SHIFT RIGHT execute instructions

denomination	inst. no.	instruction code					mnemonic	condition for fulfilment
		INS_0 (LSB)	INS_1	INS_2	INS_3	INS_4 (MSB)		
SHIFT LEFT	4	0	0	1	0	0	SHFTL	1
SHIFT RIGHT	5	1	0	1	0	0	SHFTR	1



The four-bit SHIFT LEFT and SHIFT RIGHT execute instructions are for assembling software shift registers and ring counters. They will become active on condition '1'; see the above table. Shift operations are carried out by the five-bit shift register in the data processor; the fifth cell in this shift register acts as the overflow register.

SHFTL *Instruction no. 4. This instruction causes the data of the four-bit scratchpad place addressed in the program word* (see the above figure) *to be transferred to the shift register for shift operation, the shifted data being re-entered into the same scratchpad place; upon initiation of the first in a series of SHFTL instructions the overflow register will be reset to state '0'.* After shift the data are loaded into the addressed scratchpad place with the previous contents of the overflow register being stored in location .0, the bits originally stored in locations .0, .1 and .2 moving to the *next higher* locations and the bit originally stored in location .3 staying in the overflow register. In a subsequent SHFTL instruction the bit left in the overflow register will move to location .0 of the scratchpad place addressed in that instruction.

SHFTR *Instruction no. 5. This instruction causes the data of the four-bit scratchpad place addressed in the program word* (see the above figure) *to be transferred to the shift register for shift operation, the shifted data being re-entered into the same scratchpad place; upon initiation of the first in a series of SHFTR instructions the overflow register will be reset to state '0'.* After shift, the data are loaded into the addressed scratchpad place with the previous contents of the overflow register being stored in location .3, the bits originally stored in locations .3, .2 and .1 moving to the *next lower* locations, and the bit originally stored in location .0 staying in the overflow register. In a subsequent SHFTR instruction the bit left in the overflow register will move to location .3 of the scratchpad place chosen in that instruction.

Figure 3-24 shows what happens during shift action. It is seen that the shift register acts as a circular shift register. The contents of the overflow register are retained for the next SHIFT instruction. Note that in the addressed scratchpad place the data bits will move to *higher* locations for the SHFTL instruction and to *lower* locations for the SHFTR instruction (see the definitions).

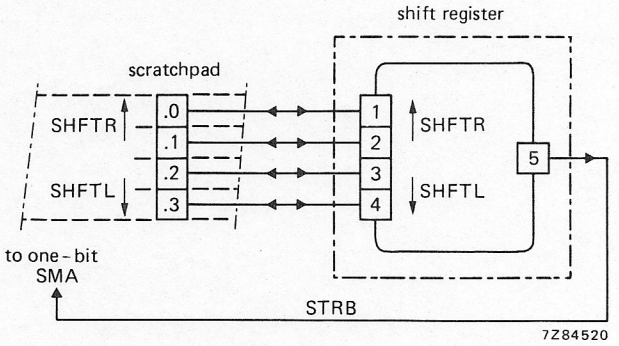


Fig. 3-24 Illustration of shift action. STRB instruction immediately following SHIFT instruction(s) extracts contents of overflow register (cell 5).

A STRB instruction *immediately following* one or more SHIFT instructions will extract the contents of the overflow register for storage in the one-bit scratchpad place selected by the STRB instruction (see Fig. 3-24); the contents of the overflow register will remain after execution of the STRB instruction. A ring counter is obtained when the STRB instruction refers to a one-bit location in the scratchpad place addressed by the first in a series of SHIFT instructions. This is illustrated in the following example.

Example 3-15 Fourteen-bit forward ring counter.

```
line   prog. word
 :       :
1010  AND     0007.0 ┐
1011  TRIG    0304.0 ┘  When input 0007.0 goes HIGH,
1012  SHFTL   0305   ┐
1013  SHFTL   0306   │
1014  SHFTL   0307   ├  shift one bit to the left
1015  SHFTL   0308   ┘
1016  STRB    0305.2    and carry back fourteenth bit.
 :       :
```

The operation of the program is clear from the description of the SHFTL instruction; the STRB instruction will carry the bit in the overflow register back to location .2 of SMA 0305 addressed in the first SHFTL instruction, thus completing the forward ring counter. Consecutive scratch-pad places (as shown in the example program) need not be chosen. The TRIG instruction prevents racing. See Fig. 3-25.

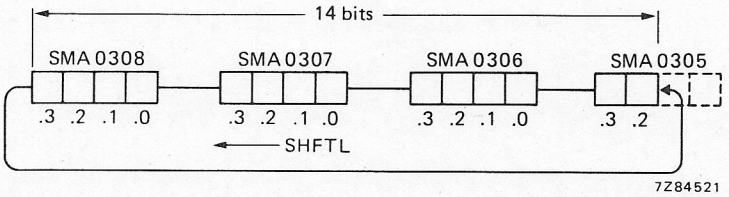
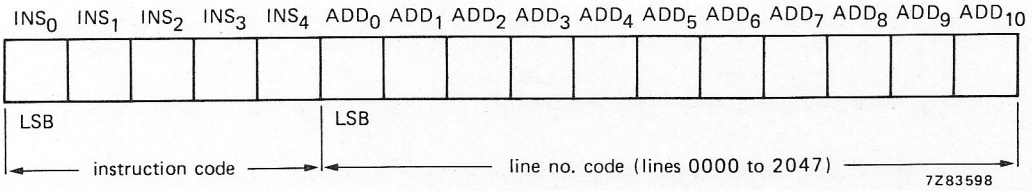


Fig. 3-25 Example 3-15: Fourteen-bit software forward ring counter.

3.2.11 The JUMP-TO-SUBROUTINE program instructions

denomination	inst. no.	instruction code					mnemonic	condition for fulfilment
		INS_0 (LSB)	INS_1	INS_2	INS_3	INS_4 (MSB)		
JUMP TO SUBROUTINE ABSOLUTE FALSE	24	0	0	0	1	1	JSAF	0
JUMP TO SUBROUTINE ABSOLUTE TRUE	25	1	0	0	1	1	JSAT	1



The JUMP-TO-SUBROUTINE instructions are amongst those that lend flexibility to the PC20 program to satisfy specific requirements of the user.

There are two types of JUMP-TO-SUBROUTINE instruction: the JUMP-TO-SUBROUTINE ABSOLUTE FALSE (JSAF) instruction, which is executed on condition 'false' (value '0'); and the JUMP-TO-SUBROUTINE ABSOLUTE TRUE (JSAT) instruction, which is executed on condition 'true' (value '1'). Their execution will generally initiate a subroutine program. They are called 'absolute' as their fulfilment causes a jump to the line number specified in the program word (beginning of subroutine program).

JSAF *Instruction no. 24. This instruction causes a jump in the execution of the program to the line number (start of subroutine) specified in the program word* (see the above figure) *when the condition is 'false'.*

Since the address part in the program word contains not more than eleven bits, the highest line number that can be specified is 2047. *So, if an MM20 or MM21 program memory is used (8k memory capacity), it is good practice to program the subroutines on the lower line numbers.*

For condition '0', the JSAF instruction becomes active and causes:

- the line number of the JSAF instruction to be loaded into the jump register; this will ensure continuation of the program in which the JSAF instruction appears once the subroutine has been carried out.
- the condition '0' to be loaded into the jump register; this condition can be used upon return from the subroutine(s) for any execute instruction(s) written immediately behind the JSAF instruction.
- the line number specified in the JSAF-program word to be loaded into the address counter; consequently, this will become the next line number in the execution of the program (beginning of subroutine).

For condition '1' the JSAF instruction will not become active and so the jump register will not be loaded. The function of the jump register is explained in Section 3.2.12: The RETURN program instruction.

JSAT *Instruction no. 25. This instruction causes a jump in the execution of the program to the line number (start of subroutine) specified in the program word* (see the figure at the head of this section) *when the condition is 'true'.*

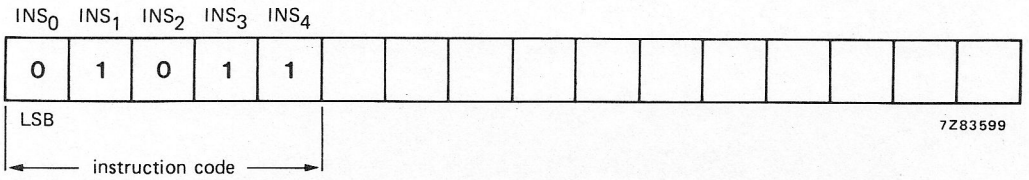
Here too, the highest line number at which a subroutine can start is 2047.

An active JSAT instruction will have the same effect as the active JSAF instruction but now condition '1' will be stored in the jump register. Examples using the JSAT instruction are given in Section 3.2.16.

The JSAT or JSAF instruction does not always initiate a subroutine program; see Example 3-17 where a RETURN instruction is not included.

3.2.12 The RETURN program instruction

denomination	inst. no.	instruction code					mnemonic	condition for fulfilment
		INS_0 (LSB)	INS_1	INS_2	INS_3	INS_4 (MSB)		
RETURN	26	0	1	0	1	1	RET	0 or 1



The RETURN program instruction is unconditional as it will be executed regardless of the value of the condition; see the table. It is written on the last line number of the subroutine program for return to the program containing the JUMP-TO-SUBROUTINE instruction. Because the line number of the JUMP-TO-SUBROUTINE instruction is stored in the jump register (see foregoing section) it is not necessary for the program word to contain an address part; see the figure.

RET *Instruction no. 26. This instruction ends the subroutine and returns the program to the line number following that on which the JSAT or JSAF instruction is written.*

When the RET program instruction is met (end of subroutine), the following happens:

- The line number previously loaded into the jump register is transferred to the address counter, which will make one step to obtain the next line number for execution of the program, i.e. that following the line number on which the JUMP-TO-SUBROUTINE instruction appears.
- The condition previously loaded into the jump register is transferred to the logic analyser; this will become the condition for any execute instruction(s) immediately following the JUMP-TO-SUBROUTINE instruction.

It is clear that the condition valid for the JUMP-TO-SUBROUTINE program instruction will become the condition for any subsequent execute instruction(s) in the same instruction block either directly (JUMP-TO-SUBROUTINE instruction not executed) or indirectly (JUMP-TO-SUBROUTINE instruction executed).

The jump register can contain up to four words of fourteen bits each. These fourteen bits are used as follows:

- line number, thirteen bits;
- condition, one bit (state '0' or '1').

The information stored *last* in the jump register as a result of an active JSAT or JSAF program instruction will be extracted *first* when a RET instruction is met *(Last-In-First-Out register).* The four-word organization of the jump register allows subroutine programs to be nested four deep, as Fig. 3-26 shows. More nests of subroutines can be programmed (see the dashed lines).

Whereas a subroutine cannot start on a line number higher than 2047 (see Section 3.2.11), the capacity of the jump register (thirteen line number bits) is adequate to allow return to any line number in the program (max. 8k program memory capacity).

Example 3-19 of Section 3.2.16 uses the RET program instruction together with the JSAT program instruction.

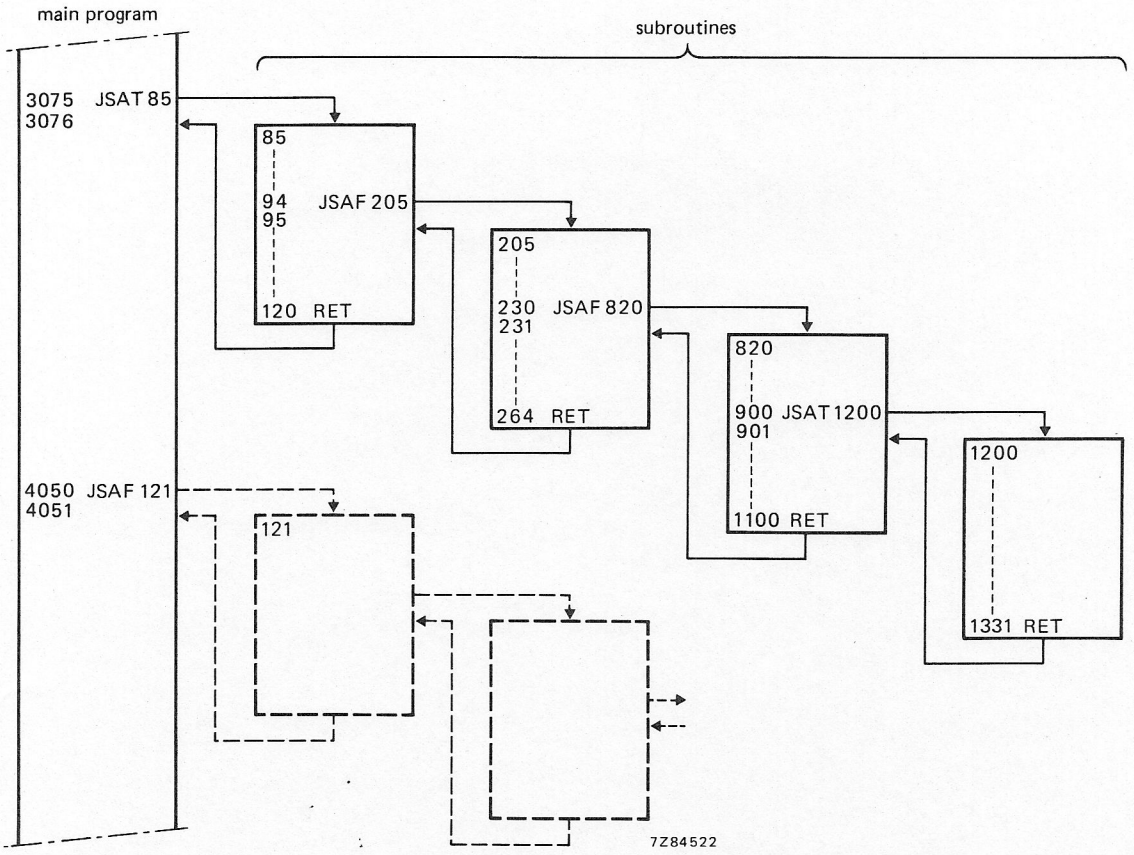
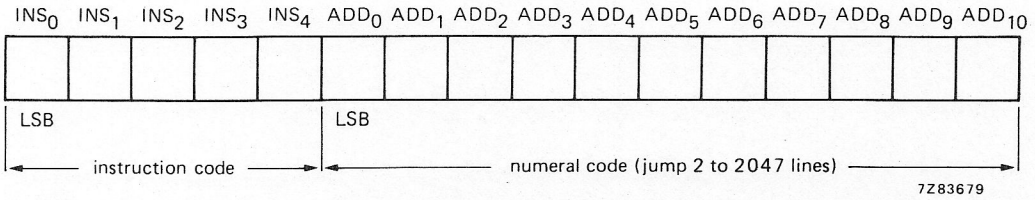


Fig. 3-26 Execution of program containing subroutines; up to four subroutines can be nested. Dashed portion shows second nest of subroutines.

3.2.13 The RELATIVE JUMP program instructions

denomination	inst. no.	instruction code					mnemonic	condition for fulfilment
		INS_0 (LSB)	INS_1	INS_2	INS_3	INS_4 (MSB)		
JUMP BACKWARD RELATIVE FALSE	29	1	0	1	1	1	JBRF	0
JUMP FORWARD RELATIVE FALSE	30	0	1	1	1	1	JFRF	0



The RELATIVE JUMP program instructions become effective on condition 'false' (value '0'). As the table shows, there are two types: the JUMP BACKWARD RELATIVE FALSE (JBRF) and the JUMP FORWARD RELATIVE FALSE (JFRF) instruction. The former is useful for programming sequencers (see Chapter 5); the latter is used to skip program portions upon request or to make branches in the program. These instructions are called 'relative' as their fulfilment causes a jump to a line number that is related to the line of program where such an instruction appears.

JBRF *Instruction no. 29. This instruction causes a jump backward in the execution of the program that is equal to the number of lines specified in the program word* (see the above figure) *when the condition is 'false'.*

JFRF *Instruction no. 30. This instruction causes a jump forward in the execution of the program that is equal to the number of lines specified in the program word* (see the above figure) *when the condition is 'false'.*

In the address processor, the number of lines specified in the program word of the RELATIVE JUMP instruction is subtracted from (JBRF instruction) or added to (JFRF instruction) the contents of the address counter (line number of RELATIVE JUMP instruction) to find the new line number where the program is to continue.

Since there are not more than eleven bits available in the program word of the JFRF and JBRF instructions (see the figure at the head of this section), the maximum jump that can be made is 2047 lines.

The following example illustrates the use of the JFRF instruction.

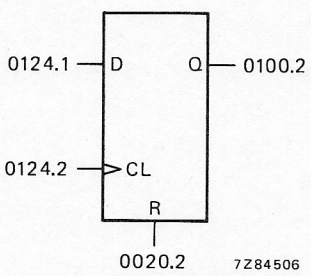


Fig. 3-27 Example 3-16: Edge-triggered D flip-flop with reset input.

Example 3-16 Edge-triggered D flip-flop with reset input.

```
line   prog. word
 :       :
0341   AND    0124.2 ┐
0342   TRIG   0079.2 ┘ When clock input 0124.2 goes HIGH,
0343   JFRF   3        (no jump)
0344   AND    0124.1   and input 0124.1 is LOW (HIGH),
0345   EQL    0100.2   output 0100.2 will become LOW (HIGH).

0346   AND    0020.2   If reset input 0020.2 is HIGH,
0347   SET 0  0100.2   output 0100.2 will become LOW.
 :       :
```

This program contains two instruction blocks, one for clocking the input into the output, the other for resetting the output. Figure 3-27 shows its hardware equivalent. In the first instruction block, the JFRF instruction will always be executed (jump to line 343 + 3 = 346) except in the PC20 cycle that the clock input goes HIGH, when the condition will become '1' (Section 3.2.4). Then, lines 344 and 345 will be read by the central processor and output 0100.2 will assume the value of the function on input 0124.1.

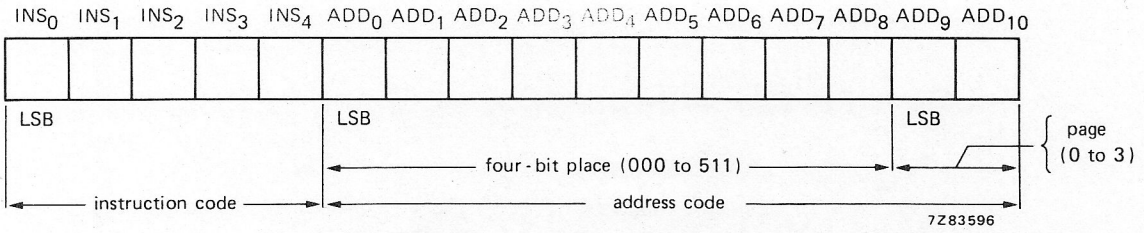
If reset input 0020.2 becomes HIGH (see second instruction block) output 0100.2 will be reset into its LOW state. The truth table of the D flip-flop programmed here is as follows:

D	CL	R	Q
X	X	H	L
X	L*	L	no change
L	L→H**	L	L
H	L→H**	L	H
X	H*	L	no change
X	H→L*	L	no change

X = don't care.
* JFRF active.
** JFRF not active.

3.2.14 The LAST INPUT OR OUTPUT ADDRESS and END program instructions

denomination	instr. no.	instruction code					mnemonic	condition for fulfilment
		INS_0 (LSB)	INS_1	INS_2	INS_3	INS_4 (MSB)		
LAST INPUT OR OUTPUT ADDRESS	31	1	1	1	1	1	LSTIO	0 or 1
END	27	1	1	0	1	1	END	0 or 1



The LAST INPUT OR OUTPUT ADDRESS (LSTIO) and END program instructions are used as a pair in the program to terminate the data processing phase. They are unconditional (see the table) and will, consequently, always be executed. These instructions will also fix the extent of the I/O phase, as will be clear from the definitions given below.

LSTIO *Instruction no. 31. This instruction causes the scratchpad address specified in the program word* (see the above figure) *to be transferred to the buffer in the address comparator to become the last input or output address in the subsequent I/O phase.*

END *Instruction no. 27. This instruction ends the data processing phase and initiates the I/O phase; it causes the scratchpad address specified in the program word* (see the above figure) *to be transferred to the address counter to become the first input or output address in the subsequent I/O phase.* During the I/O phase, the address counter starts at that address (position) and steps through its positions until the position is reached that accords with the information stored in the buffer of the address comparator as a result of the preceding LSTIO instruction; here, the I/O phase will terminate. *The END instruction must always be included in the program; if not, the PC20 system will not come out of the data processing phase unless it is reset or its supply interrupted.*

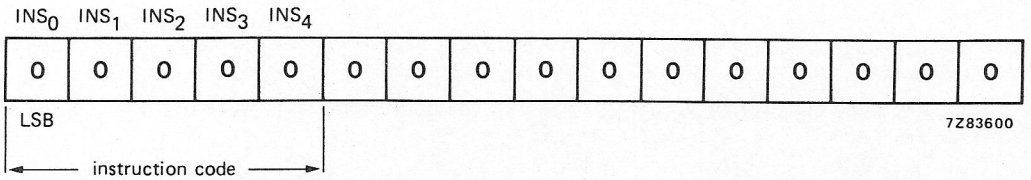
The END instruction forces the condition (state of result register) into the '0' state; this will become the condition at the start of the next data processing phase.

Owing to the facility of programming the first as well as the last input/output address, the I/O phase will not become longer than strictly necessary; thus, speediest system response will be ensured.

Examples illustrating the use of the LSTIO and END program instructions will follow in Section 3.2.16.

3.2.15 The NO-OPERATION program instruction

denomination	instr. no.	instruction code					mnemonic	condition for fulfilment
		INS_0 (LSB)	INS_1	INS_2	INS_3	INS_4 (MSB)		
NO-OPERATION	0	0	0	0	0	0	NOP	0 or 1



The NO-OPERATION (NOP) program instruction is unconditional (see the table) and serves to make spare lines in a program under test. It is indispensable for successful performance of insert and delete operations during editing (see Sections 4.3.3 and 4.3.4).

NOP *Instruction no. 0. This is a void instruction used to make empty spaces in the program.*

3.2.16 Program frame examples

A diagram of a basic PC20 program is presented in Fig. 3-5, Section 3.1. Because this program is so simple, undesirable switch-on phenomena can occur under certain circumstances. Immediately after switch-on or reset, the PC20 system will start with the data processing phase. (The UDC and reset phases are interposed upon request only – see Chapter 2.) This means that an I/O phase has not yet occurred so that data have not yet been entered into the scratchpad. Since the contents of the scratchpad are therefore arbitrary, the results of the first data processing phase will be erratic.

Program frame examples that follow show how to avoid this. Bear in mind that when the PC20 system enters its first operating cycle after switch-on or reset, the condition will be set to state '1'; when the END program instruction is met, the condition will be forced to state '0'. This is the underlying principle of the examples given here.

Example 3-17 Program frame using JSAT program instruction for start-up.

```
line   prog. word
0000   JSAT     985     Upon switch-on or reset (condition '1') jump to line 985.
0001   AND      ...  ┐
  :      :        :  ├  Program.
  :      :        :  ┘
0985   LSTIO    0023    Last I/O address in I/O phase is SMA 0023.
0986   END      0004    Terminate data proc. phase and start I/O phase with SMA 0004.
```

Figure 3-28 illustrates the execution of this program together with the values of the condition valid for line 0. The JSAT program instruction written at the beginning of the program will only be executed in the first PC20 cycle after switch-on or reset because the data processing phase in that cycle opens with condition '1'. It causes a jump to lines 985 and 986 where the LSTIO and END instructions have been written. So, an I/O phase will immediately occur to enter data into the scratchpad before the program is executed. At the end of each data processing phase the condition is reset to state '0' by the END instruction; consequently, in all PC20 cycles except the first, the data processing phase will be entered under condition '0', the JSAT instruction will not be active and the program on lines 1 to 984 terminated with the LSTIO and END instructions will be executed.

The program must not exceed 2k length because a line number higher than 2047 cannot be described in the program word of the JSAT instruction (Section 3.2.11).

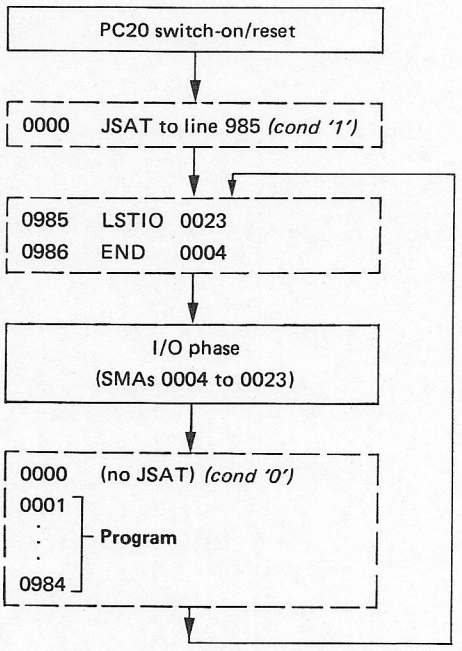


Fig. 3-28 Example 3-17: Execution of program. Dashed boxes indicate that PC20 is in data processing phase; line numbers specified in left-hand part of boxes.

Example 3-18 Program frame using JFRF program instruction for start-up.

```
line   prog. word
0000   JFRF      3      Upon switch-on or reset (condition '1'), i.e. no jump,
0001   LSTIO     0031   ┐
0002   END       0012   ┘ go to I/O phase for start-up.
0003   ANDNT     ...    ┐
 :      :               ├ Program.
 :      :               ┘
2592   LSTIO     0047   ┐
2593   END       0004   ┘ Go to I/O phase – normal.
```

The execution of this program is as shown in Fig. 3-29, together with the values of the condition valid for a given line number. Here, the jump (JFRF) program instruction will always be executed (jump to line 0 + 3 = 3) except in the first PC20 cycle after switch-on or reset, when the condition is '1'. Then, the I/O phase follows after lines 1 and 2 for start-up before the program on lines 3 to 2593 is executed. Note, the extent of this I/O phase can be made to be different from that of the I/O phase for normal operation. This program has the following benefits:

- It can have any length up to 8k.
- An I/O phase can be programmed for start-up conditions in addition to the I/O phase for normal operation.

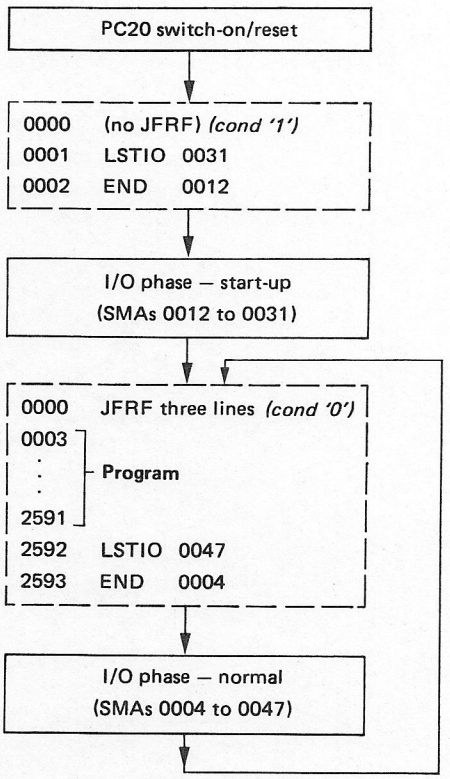


Fig. 3-29 Example 3-18: Execution of program. Dashed boxes indicate that PC20 is in data processing phase; line numbers specified in left-hand part of boxes.

Example 3-19 Program frame with intermediate I/O phase.

line	prog. word		
0000	JSAT	6	Jump to line 6 in first cycle.
0001	RET		Common return for I/O subroutines.
0002	LSTIO	0031	I/O subroutine (start-up).
0003	END	0012	
0004	LSTIO	0203	I/O subroutine (normal).
0005	END	0004	
0006	JSAT	2	Jump to I/O subroutine (start-up) in first cycle.
0007	ORNT	...	*First program part.*
:	:		
1911	EQL	...	
1912	AND	0000.1*	Unconditional jump to I/O subroutine (normal).
1913	JSAT	4	
1914	AND	...	*Second program part.*
:	:		
4377	STRB	...	
4378	AND	0000.1*	Unconditional jump to I/O subroutine (normal).
4379	JSAT	4	
4380	JSAT	7	Jump to first part of program.

* Constant '1' (Table 3-3, Section 3.2).

The program will be executed as shown in Fig. 3-30, which also shows the values of the condition valid for a given line number. To understand program operation it must be called to mind (Section 3.2.12) that, when the RET program instruction is met, the new line number will become equal to:

(line number last loaded by a JSAT instruction into jump register) + 1.

There are two I/O subroutines in the program: start-up and normal. Each subroutine is initiated by a JSAT instruction and terminated by the common RET instruction on line 1. The start-up I/O subroutine will occur once only as the JSAT instructions on line 0 and 6 are obeyed only in the first data processing phase after switch-on or reset (condition '1' prevailing). The I/O subroutine for normal operation will always be carried out because of the "unconditional jumps" on lines 1912, 1913 and 4378, 4379 (SMA 000.1 constant '1'). On line 4380, bottom of Fig. 3-30, the JSAT instruction will be executed for return to the first part of the program as on the previously executed line number, 0001, condition '1' has been transferred from the jump register to the logic analyser.

It is seen that the program is executed in two parts (interposition of an I/O phase). Thus, the I/O phases will occur at an increased repetition rate. This program is useful, for instance, for fast counting of objects with the count program appearing both in the first and second program part.

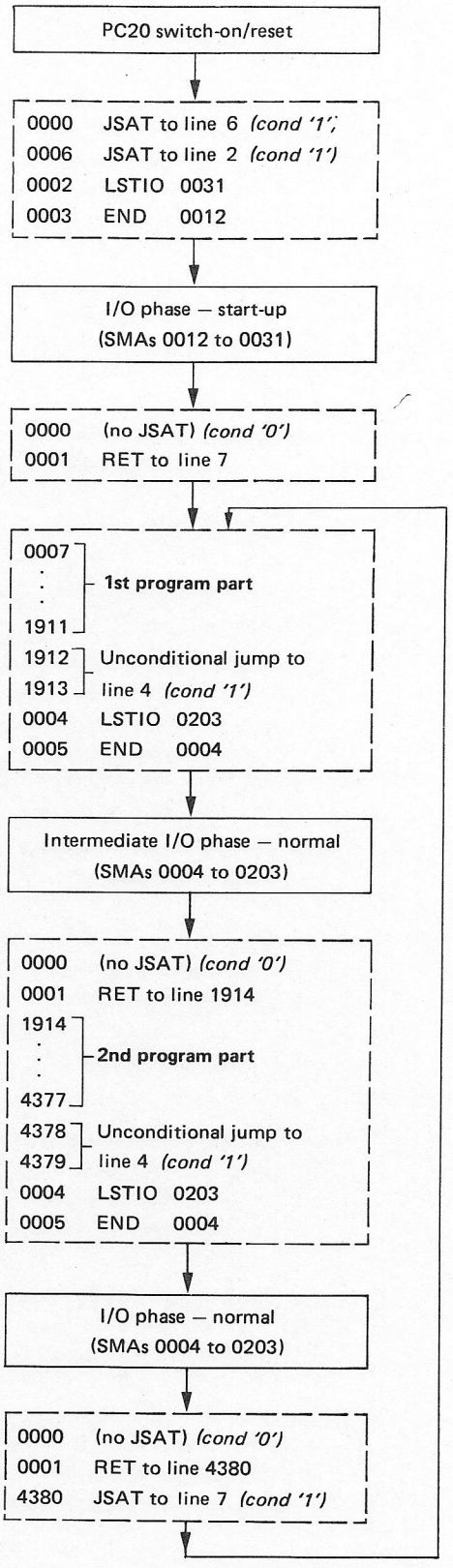


Fig. 3-30 Example 3-19: Execution of program. Dashed boxes indicate that PC20 system is in data processing phase; line numbers specified in left-hand part of boxes.

3.3 Cycle time and instruction time

The *PC20 cycle time* depends on various factors:

- Program length.
- Whether or not jumps occur in the program.
- Instruction time (worked out below).
- Length of I/O phase. See Table 3-13. The duration of the I/O phase is equal to the specified figure times the number of I/O addresses scanned.

TABLE 3-13 Finding duration of I/O phase.

configuration	time per scratchpad memory address (four bits) in μs
CP21	2,65
CP20, CP24	3,14
CP22 with MM20, MM21 or MM22	3,35

Note: Mentioned times are for the I/O-modules in the first (main) rack. Per additional (extension) rack 0,21 μs must be added.

The factors governing the length of the *instruction time* (time spent on a given line of program) will be considered in greater detail here; the following points apply.

1. *Instruction executed/not executed.* Conditional instructions are executed when the condition required for their fulfilment has been satisfied. If not, the instruction will not be obeyed and a shorter instruction time will result.
2. *Write-in/no write-in.* For the one-bit instructions – TRIG, EQL, EQLNT, SET 1, SET 0 and STRB – an unnecessary write operation is prevented, that is, if the bit to be written in has the same value as the bit already stored. Also here, the instruction time will become shorter.
3. *Count/no count.* Any COUNT instruction in a string of these instructions will not be executed, even when the condition for execution is met, unless a "carry" or "borrow" occurs in the preceding instruction. The instruction time will be shorter when the instruction is not carried out.
4. *STRB and STRD instructions.* The instruction time for the STRB and STRD instructions will vary widely when these instructions are used to carry out an arithmetic operation (STRB or STRD instruction immediately following ADD, SUBTR, MULT or DIV instruction).

The central processor will therefore step through the program with minimum delay, thus minimizing the time for data processing.

Instruction times are compiled in Tables 3-14 and 3-15. It is seen that the CP21 yields the fastest response.

TABLE 3-14 Instruction time for execute instructions: condition for execution not satisfied.

configuration	instruction time in μs
CP21	0,84
CP20, CP24	1,26
CP22 with MM20, MM21 or MM22	1,46

As the PC20 cycle time varies during system operation to ensure the speediest response, it can not be relied upon for timing. Crystal-controlled clock signals are available for that purpose in the scratchpad; see Table 3-16.

TABLE 3-16 Internal SMAs providing clock pulses (1:1 mark/space ratio).

repetition time (s)	SMA
0,01	0000,3
0,1	0001,0
1	0001,1
10	0001,2
60	0001,3

The accuracy of timing depends not so much on the accuracy of the clock as on the instant that the clock pulse is sensed during data processing. The slower the clock the greater the accuracy that can be expected; see Section 5.10.

TABLE 3-15 Instruction times in µs. For execute instructions: condition for execution assumed to be satisfied. See also Table 3-14.

inst. no.	mnem.		configuration CP21	CP20, CP24	CP22 with MM20, MM21 or MM22
0	NOP		0,84	1,26	1,46
1	TRIG	no bit change[a]	0,84	1,26	1,46
		bit change[a]	1,40	2,09	2,30
2	EQL	no bit change[a]	0,84	1,26	1,46
3	EQLNT	bit change[a]	1,40	2,09	2,30
4	SHFTL		1,26	1,88	2,30
5	SHFTR		1,67	2,51	2,92
6	CNTD	no execution in string[b]	0,84	1,26	1,46
7	CNTU	execution in string[b]	1,40	2,09	2,51
8	SET 0	no bit change[a]	0,84	1,67	1,88
9	SET 1	bit change[a]	1,40	2,09	2,30
10	STRB	no bit change[a]	1,12	1,67	1,88
		bit change[a]	1,67	2,09	2,30
11	FTCHB		1,40	2,09	2,09
12	FTCHC		0,84	1,26	1,46
13	FTCHD		0,84	1,26	1,46
14	STRD	after FTCHC, FTCHD, COMP, CNTD/U & SHFTL/R	1,26	1,88	2,30
		after ADD & SUBTR	4,89	7,11	7,32
		after MULT 9999 × 1	164[c]	245[c]	245[c]
		after DIV 9999 ÷ 1	236[c]	345[c]	345[c]
15	COMP		0,84	1,26	1,46
16	AND		0,84	1,26	1,46
17	ANDNT		0,84	1,26	1,46
18	OR		0,84	1,26	1,46
19	ORNT		0,84	1,26	1,46
20	ADD		0,84	1,26	1,46
21	SUBTR		0,84	1,26	1,46
22	MULT		0,84	1,26	1,46
23	DIV		0,84	1,26	1,46
24	JSAF		0,98	1,46	1,67
25	JSAT		0,98	1,46	1,67
26	RET		0,98	1,46	1,67
27	END		0,84	1,26	1,46
29	JBRF		0,98	1,46	1,67
30	JFRF		0,98	1,46	1,67
31	LSTIO		0,84	1,26	1,46

[a] See point 2 page 3.39. [b] See point 3 page 3.39. [c] Maximum value.

4 The PU20 facilities

Contents

+ TRANSFERT PU20 → MX80

4 THE PU20 FACILITIES

Together with the facilities of the RAM program memory in the PC20, it is the PU20 programming unit that gives the PC20 programmable controller its flexibility; the PU20 communicates with the PC20 system via the PU21 interface module. In contrast with hard-wired systems, program changes are readily made using the PU20 programming unit and either a CP21 central processor or a CP22 central processor together with the MM21 memory module. A key switch is used to protect all programming facilities; without the key, the contents of the program memory and the scratchpad memory can be monitored only.

It should be noted that, after programming, the PC20 system can work with the programming unit disconnected.

4.1 The PU20 functions

The mains-powered desk-top programming unit PU20 is portable and communicates with the PC20 via a rack-mounted interface module PU21. One PU20 can program any number of PC20 systems. The PC20 cable is plugged into the connector on the interface front panel. Because the PC20 must be switched off to remove or insert an interface module, it is recommended that the PU21 is left in place as long as program changes may be necessary, so that the PU20 can be connected and disconnected without disrupting the operation of the PC20.

Fig. 4-1 shows the PU20 programming unit (440 mm width, 142 mm height, 295 mm depth). Inputs and outputs are the cable that plugs into the PU21 interface, two EPROM sockets located under a hinged cover in the upswept part of the PU20, an RS449/423 connector and a DIN audio cassette connector at the rear of the PU20.

The PU20 communicates to the user through a sixteen-digit LED display. Commands to the PU20 are given via the keyboard.

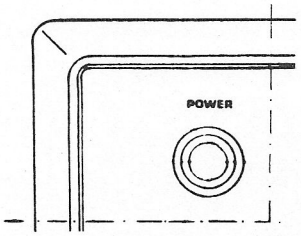
The PU20 permits programming, program changes and monitoring, also program dumping and loading.

All PU20 functions will now be discussed in greater detail.

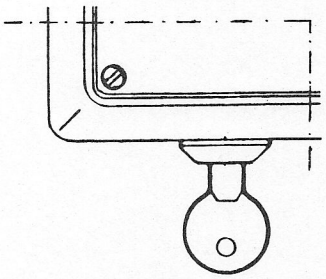
Fig. 4-1 Desk-top PU20 programming unit.

4.1.1 Switch functions

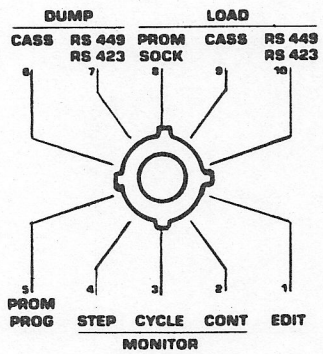
Mains switch. This switch is a push-button type with power on/off indication. Upon switch-on, all PU20 functions are initialized or reset.



Key switch. With the key switch off, only monitoring is possible. When the key switch is turned on, all other PU20 functions will be activated. In the ON-position, the key is retained in its lock.



Mode selector switch. With the mode selector switch one of ten modes is chosen:



MODE 1: EDIT MODE. With the key switch on, programming and program changes can take place. The PC20 is halted in the UDC (UpDate and Check) phase and its C-MOS RAM (CP21 or MM21) will be accessible for programming. Also program words can be deleted or new words inserted.

MODE 2: MONITORING 'CONTINUOUS'. In this mode, the PC20 is in its running condition. With the key switch *off*, it is only possible to monitor program performance. By monitoring instructions and the resulting changes in scratch-pad data it can be seen if program functions are performed correctly. With the key switch *on*, program words and scratchpad data can be changed; it should be borne in mind that such changes will have immediate consequence for the program, which is in active condition here.

MODE 3: MONITORING CYCLE BY CYCLE. In this mode (key switch on) the PC20 will carry out single cycles on command, starting in the UDC phase. This is useful to check single-event phenomena. If required, changes in program words and scratchpad data can be made.

MODE 4: MONITORING STEP BY STEP. This is another useful mode for fault finding (key switch on). The PC20 is made to step through its date processing phase so that each instruction (logic as well as execute) can be individually checked. When the end of program is reached, the PC20 will pass through the subsequent I/O (input/output) phase and wait at the start (line no. 0000) of the next processing phase.

MODE 5: EPROM PROGRAMMING. In this mode (key switch on) EPROMs placed in the sockets on the PU20 can be programmed (all or part of the PC20 program can be dumped). Verification occurs while programming proceeds.

MODE 6: DUMP ON CASSETTE. All or part of the PC20 program can be dumped on an audio cassette when the key switch is on.

MODE 7: DUMP ON RS449/423. All or part of the PC20 program can be dumped on teletype, VDU, data logger or other peripheral equipment fitted with the RS449/423 standard interface, when the switch key is on.

MODE 8: LOAD FROM EPROMs. All or part of the program stored in EPROMs can be loaded into the C-MOS RAM (CP21 or MM21) of the PC20 (key switch on). The EPROMs are placed in the sockets on the PU20. Verification occurs while loading proceeds.

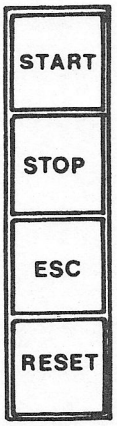
MODE 9: LOAD FROM CASSETTE. All or part of the program stored on audio cassette is loaded into the C-MOS RAM of the PC20 (key switch on). This mode is also used to *verify* the program just loaded into the PC20 program memory or previously dumped on cassette (Mode 6).

MODE 10: LOAD FROM RS449/423. All or part of the program dumped in peripheral equipment is loaded into the C-MOS RAM of the PC20 (key switch on). This mode is also used to *verify* the program just loaded into the PC20 program memory or a previously dumped program (Mode 7).

In Modes 5 to 10, parts of the program to be dumped can be selected in data blocks of 1k.

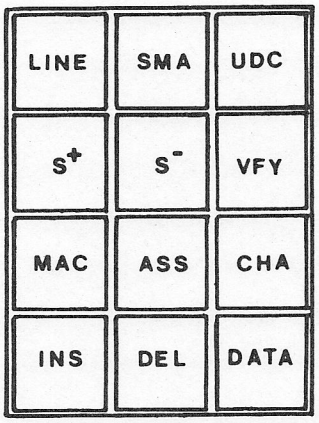
4.1.2 Keyboard functions

Control keys. These keys control the operation of the PU20. Pressing the START key initiates the selected operation. Pressing the STOP key will stop the operation in progress. The ESC(APE) key is used to proceed to another operation. Depressing the RESET key will disrupt any operation; it is solely used where operation does not occur as desired.



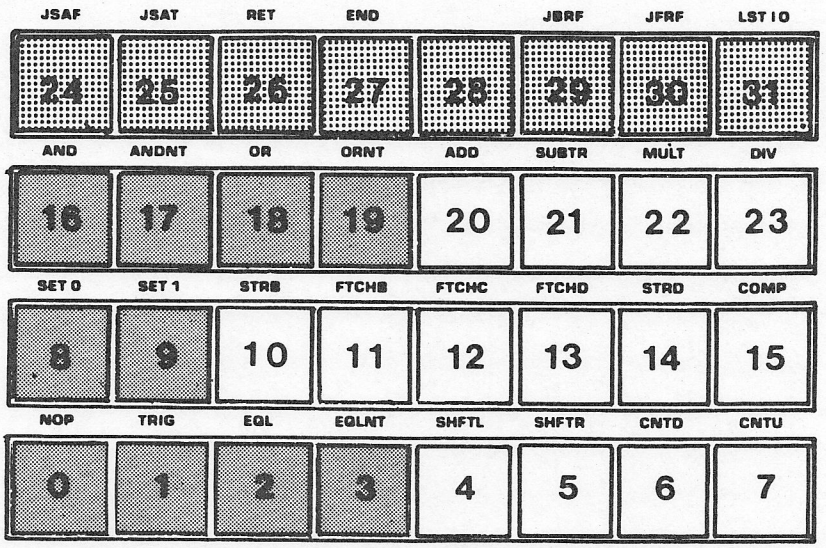
Program function keys. These keys comprise:

- 'LINE' and 'SMA' to tell the programming unit that a LINE number (program memory address) or Scratchpad Memory Address is to be entered on the digit keyboard.

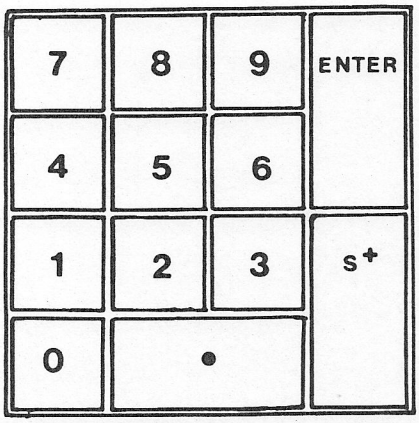


- 'UDC' to inspect a line number or the contents of a scratchpad address that is jumped over by the PC20 or that is located beyond an END instruction.
- 'S^+' and 'S^-' to advance or recede one line number (one step in the program memory).
- 'VFY' to verify program loading or a dumped program (Modes 9 and 10).
- 'MAC(RO)' to initiate programming of one of several complete software circuits selected via the digit keyboard.
- 'ASS(EMBLE)' to write the selected software circuit in the program memory after the various circuit parameters have been entered on the digit keyboard.
- 'CHA(NGE)' to change a program word in an existing program or to update the contents of the selected scratchpad address.
- 'INS(ERT)' to insert a new program word in an existing program.
- 'DEL(ETE)' to delete one or more program words (up to ninety-nine words in one operation).
- 'DATA' to update the contents of a selected scratchpad address (numbers 0 to 15); for this function, the DATA key must be pressed together with the CHANGE key. Further, the DATA key is pressed together with one of the keys on the digit keyboard to enter the length of program to be dumped or loaded (1k to 8k program length).

Program instruction keys. These keys are used to enter program instructions (see Chapter 3: "The PC20 instruction set"). Each key carries its appropriate instruction number and the instruction mnemonics are shown *above* the keys.



Digit keyboard. This keyboard is used to address both the program memory and the scratchpad memory. To address specific bit places in the scratchpad, the decimal point key must be used as well as digit keys. The digit keys can also be used to change the contents of a selected scratchpad address. Further, digit keys 0 to 9 are used to determine the length of a program to be dumped or loaded. Pressing the ENTER key will either make the existing program word visible or write the new program word in the selected program-memory address (line number). Once the line number where programming is to start has been typed in, operating the S^+ key will cause the PU20 to advance one step in the program memory to enter the next word; the function of the S^+ key is identical to that of the S^+ program function key. The ENTER key is also used to enter various parameters for a given operation.



4.1.3 Display functions

Multi-digit display. This consists of sixteen seven-segment LED displays, divided as follows:

MODE (2 digits) indicating the number of the mode selected with the mode selector switch.

LINE NUMBER (4 digits) indicating the line number (program-memory address).

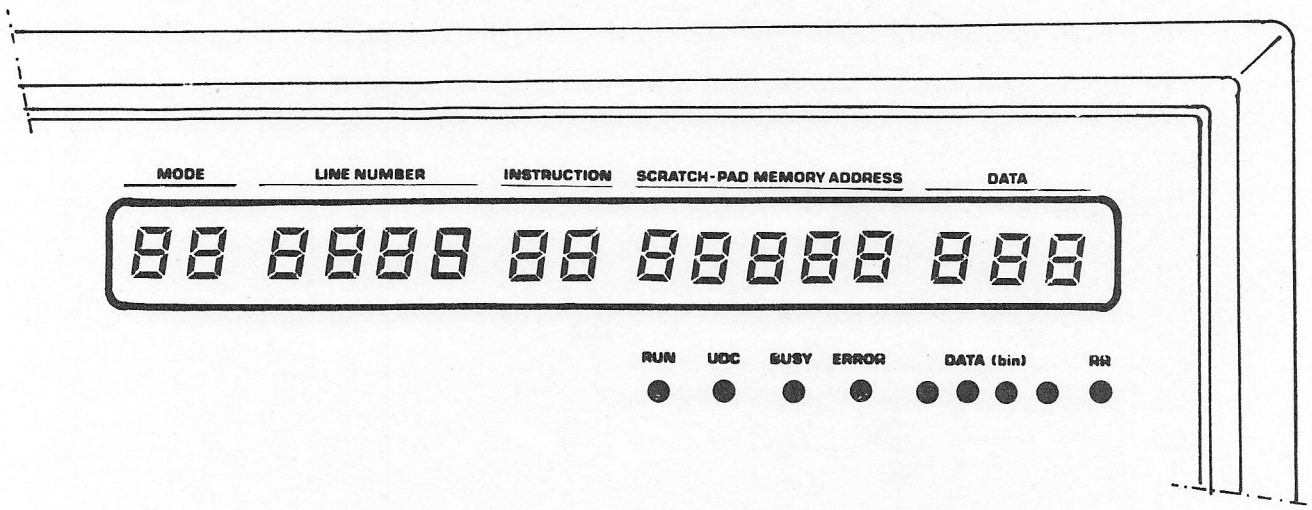
INSTRUCTION (2 digits) indicating the number of the instruction selected on the program instruction keyboard. Other purposes of this display section are explained when the various modes are discussed.

SCRATCHPAD MEMORY ADDRESS (5 digits) indicating the scratchpad address (one-bit or four-bit address); this display section is also used for other purposes, as explained further in this chapter.

DATA (3 digits) indicating the contents of the scratchpad address shown on the SCRATCHPAD MEMORY ADDRESS display. Other purposes of this display section are explained when the various modes are discussed.

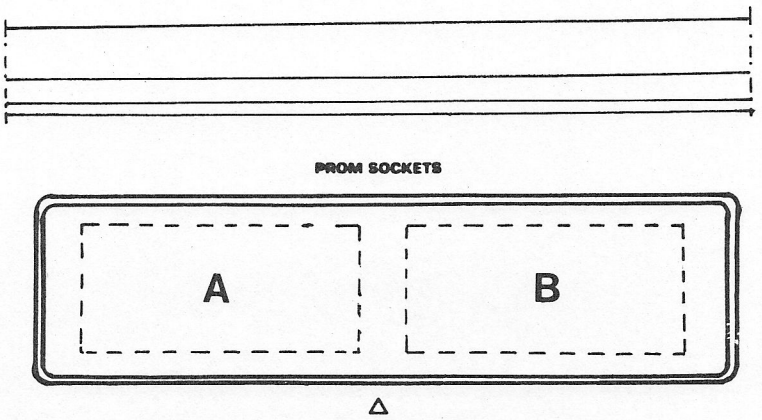
CONDITION INDICATOR LEDs. These LEDs show the operation of the PU20 or its condition. The 'RUN' LED when lit indicates that the PU20 is carrying out the specific command given on the keyboard; the 'BUSY' LED when lit indicates data transfer between the programming unit and the PC20 system or peripheral equipment. The 'UDC' LED lights when the 'UDC' key is depressed, and indicates that the PC20 is operating in the UDC phase. If a command is not understood by the PC20, the 'ERROR' LED will light.

DATA AND 'RR' LEDs. The four data LEDs – indication 'DATA(bin)' – show the contents of the scratchpad address in binary form; this corresponds to the data shown by the DATA display. The 'RR' (Result Register) LED indicates the state of the condition flip-flop while the logic instructions are being processed (monitoring modes 2, 3 and 4).



4.1.4 EPROM sockets

For program dumping or loading (Modes 5 and 8) the EPROMs are clamped in their sockets (loading only into C-MOS RAM). Two 2758 EPROMs (1k8) are used for 1k memory capacity and two 2716 EPROMs (2k8) are necessary to obtain 2k memory capacity.

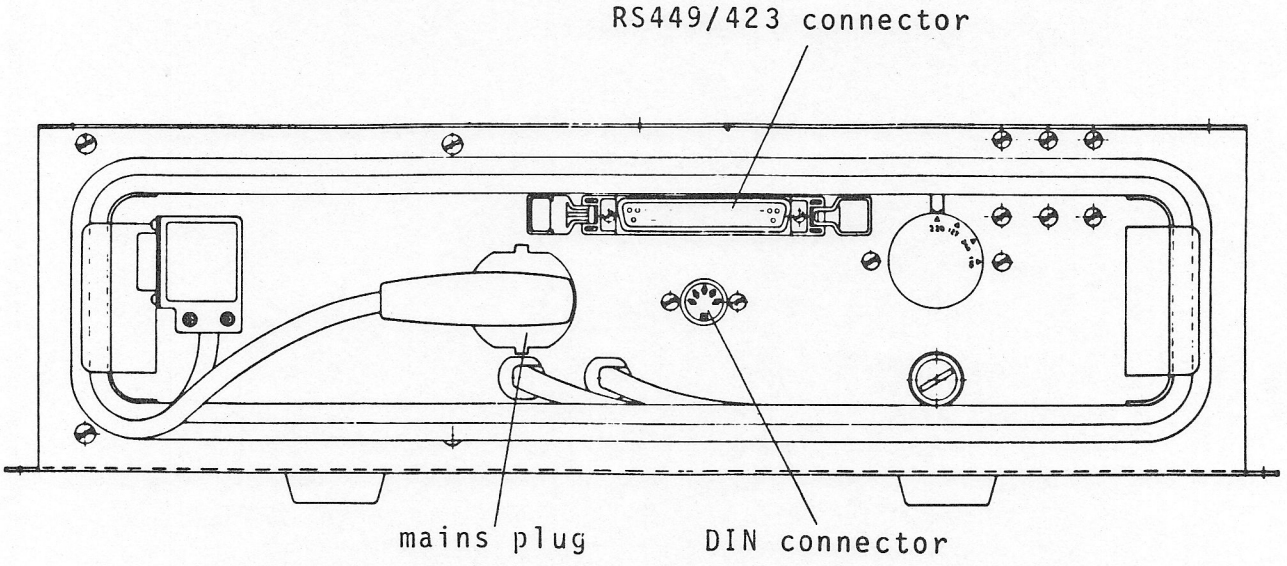


4.1.5 Coordination with peripherals

For coordination with peripherals the following connectors are available on the rear of the PU20:

DIN connector to connect an audio cassette recorder for program dumping and loading (Modes 6 and 9).

RS449/423 connector to connect peripherals, such as a teletype, VDU, data logger, etc., for program dumping and loading (Modes 7 and 10); the peripheral must be equipped with the RS449/423 standard interface.



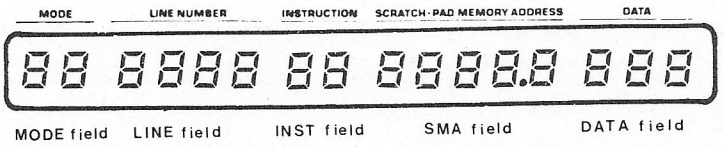


Fig. 4-2 The PU20 display

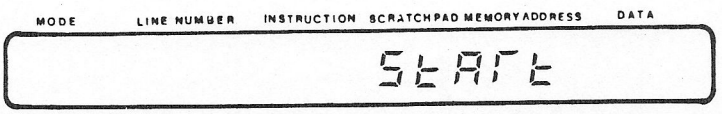


Fig. 4-3 'Start' display

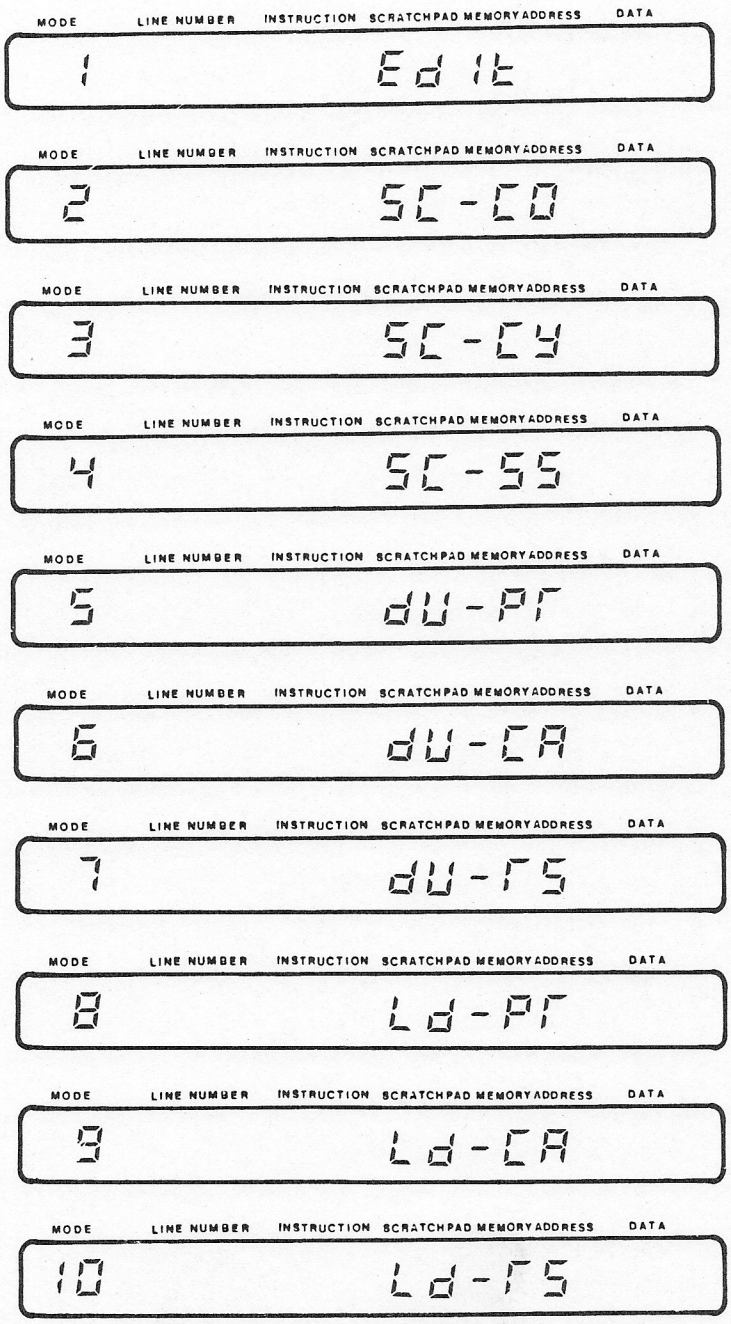


Fig. 4-4 Mode display.

4.2 Using the PU20

4.2.1 The multi-digit display

Fig. 4-2 shows the sixteen-digit seven-segment display of the PU20. The following notation will be adopted throughout the text:

- MODE field for the two-digit mode display section.
- LINE field the four-digit line-number display section.
- INST field for the two-digit instruction-number display section.
- SMA field for the five-digit scratchpad memory address display section.
- DATA for the three-digit data display section.

4.2.2 Mode selection

Mode selection occurs in the following steps:

1. Depress ESC(APE) or RESET key. This ensures that the PU20 starts in a well-defined (reset) condition. The word 'START' will now appear in the SMA field (Fig. 4-3).
2. Connect peripherals or insert EPROMs, if necessary. The EPROMs are clamped in the sockets through lever action (Section 4.1.4).
3. Select desired mode using the mode selector switch (Section 4.1.1).
4. Set key switch in ON-position, if required.
5. Press START key. Now the mode number will appear in the MODE field and the type of mode will be indicated in the SMA field. This is clearly shown in Fig. 4-4.

4.2.3 Error indication

All PU20 operations are continuously supervized to ensure that they are proceeding correctly. The user will be alerted as soon as an error occurs or if the wrong keys are depressed. An error code will then appear in the DATA field; its significance is specified in Table 4-1.

TABLE 4-1 Error code specification

display in DATA field	description
E1	wrong key depressed
E2	wrong program function key
E3	wrong bit location
E4	wrong scratchpad memory address
E5	key switch not ON
E6	wrong line number in Modes 5 to 10
E7	parity error
E8	baud rate or overrun error
E9	format error (Modes 6, 7, 9 and 10)
E10	no data present (Modes 9 and 10)
E11	EPROM program error
E12	verify error
E13	wrong format code (Mode 7)
E14	starting address of cassette or RS449/423 too high
E15	macro error

4.3 Edit mode

For the edit mode, the PC20 system must be decoupled from the controlled process. The key switch is placed in its ON-position. To permit editing, the PC20 must have a C-MOS RAM program memory (CP21 central processor or MM21 memory module). The following facilities exist:

- Entering a program
- Changing program words
- Deleting program words
- Inserting program words
- Entering macros
- Search instruction.

LINE NUMBER SELECTION. This can be done in either of the following ways:

- Starting from a given line number, depress the S^+ key n times (for n steps forward in the program memory), or depress the S^- key n times (for n steps backward), until the desired line number is obtained.
- The desired line number is obtained directly by depressing the LINE key, then entering the line number on the digit keyboard, e.g. 12 for line no. 0012.

ENTER KEY FUNCTIONS. The ENTER key has a dual function, that of read-out and write-in.

- *Read-out.* When the ENTER key is operated after a particular line number has been chosen, the program memory contents stored at that address will be displayed. This function is shown in the examples as a dashed square because here operation of the ENTER key is for monitoring purposes only.
- *Write-in.* When the ENTER key is operated after a program word (instruction part + address part) has been typed in, that program word will be stored at the selected program memory address (line number) and the existing contents removed. *It is good practice to check the typed-in program word on the display before write-in.* The stored program word is fed back to the display to allow the user to check the storage process. The write-in function of the ENTER key is shown in the examples as a solid square.

4.3.1 Entering a program

The following program part will be taken as an example.

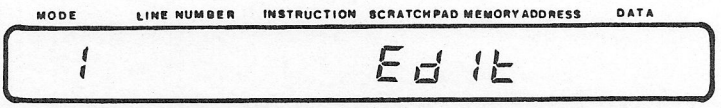
line	program word			
0000	AND	1.1	(1 s clock)	When pulse occurs
0001	TRIG	39.0	]	make one count (units).
0002	CNTU	12	]	
0003	CNTU	13		When carry occurs make one count (tens).
0004	STRB	16.0		Store counter state bit on SMA* 16.0
0005	AND	4.0		If 4.0 becomes HIGH
0006	JSAT	416		jump to line no. 416.
:	:			:

* Scratchpad Memory Address.

In the following example, the squares represent the keys to be operated.

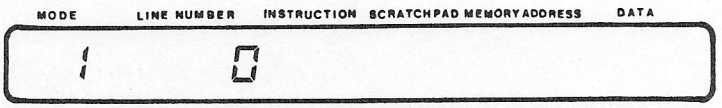
Example 4-1 Entering line nos. 0000 to 0006.

Edit mode selected:
(see Section 4.2.2)



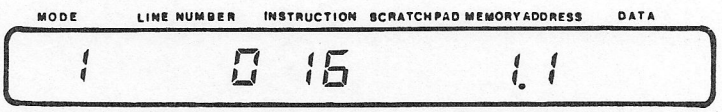
Start at line no. 0000:

[LINE] [0]



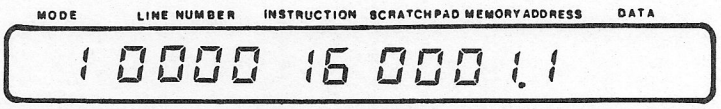
Type in program word:

[AND] [1] [.] [1]



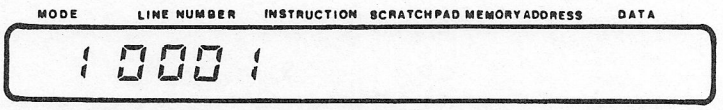
Check display, then enter into program memory:

[ENTER]



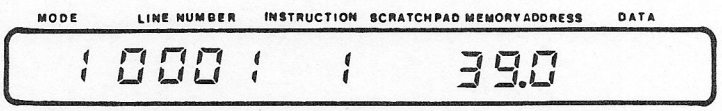
Recheck display and select next line no.:

[S+]



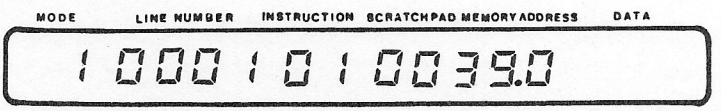
Type in 2nd program word:

[TRIG] [3] [9] [.] [0]



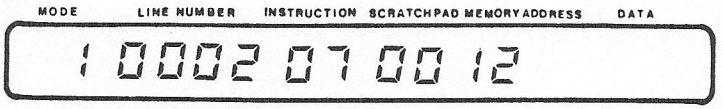
Check display, enter and recheck display:

[ENTER]



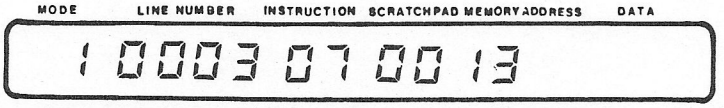
Type in 3rd program word, check display, enter and recheck display:

[S+] [CNTU] [1] [2] [ENTER]



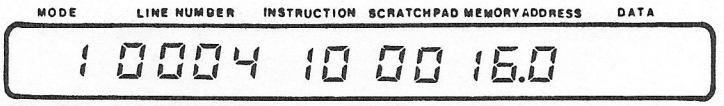
Same with 4th program word:

[S+] [CNTU] [1] [3] [ENTER]



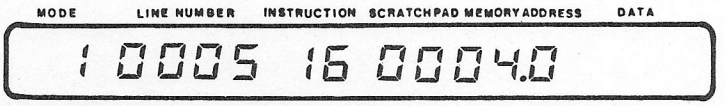
Same with 5th program word:

[S+] [STRB] [1] [6] [.] [0] [ENTER]



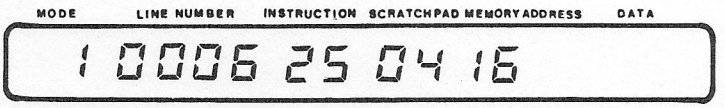
Same with 6th program word:

[S+] [AND] [4] [.] [0] [ENTER]



Same with 7th program word:

[S+] [JSAT] [4] [1] [6] [ENTER]



It will be seen that programming takes the following routine:

TYPE IN PROGRAM WORD/CHECK DISPLAY/ENTER/RECHECK DISPLAY/STEP

The 'RUN' LED will become lit as soon as a command is initiated (depressing the LINE or S^+ key). When the ENTER key is depressed for write-in, the 'BUSY' LED becomes illuminated for an instant to indicate that the typed-in program word is being transferred to the program memory and the new contents fed back to the display. Data transfer completed, both the 'RUN' and 'BUSY' LED will extinguish to indicate that the PU20 is now ready to receive a new command.

As example 4-1 shows, the number appearing in the SMA field does not always represent a scratchpad memory address. On line 0006 it shows the line number to which the jump, if active, will occur (first line number, 0416, of subroutine program). In the case of a relative jump, it shows the number of program memory addresses to retrace or advance. Table 4-2 gives a full account.

TABLE 4-2 Significance of SMA display*

inst. no.	mnemonic	SMA field shows:
00	NOP	blank
01	TRIG	one-bit SMA
02	EQL	one-bit SMA
03	EQLNT	one-bit SMA
04	SHFTL	four-bit SMA
05	SHFTR	four-bit SMA
06	CNTD	four-bit SMA
07	CNTU	four-bit SMA
08	SET 0	one-bit SMA
09	SET 1	one-bit SMA
10	STRB	one-bit SMA
11	FTCHB	one-bit SMA
12	FTCHC	number (0 to 15)
13	FTCHD	four-bit SMA
14	STRD	four-bit SMA
15	COMP	four-bit SMA
16	AND	one-bit SMA
17	ANDNT	one-bit SMA
18	OR	one-bit SMA
19	ORNT	one-bit SMA
20	ADD	four-bit SMA
21	SUBTR	four-bit SMA
22	MULT	four-bit SMA
23	DIV	four-bit SMA
24	JSAF	start line number of subroutine program
25	JSAT	start line number of subroutine program
26	RET	blank
27	END	four-bit SMA (first input or output address)
29	JBRF	no. of lines to jump backward (max. 2047 lines)
30	JFRF	no. of lines to jump forward (max. 2047 lines)
31	LSTIO	four-bit SMA (last input or output address)

* Scratchpad Memory Address.

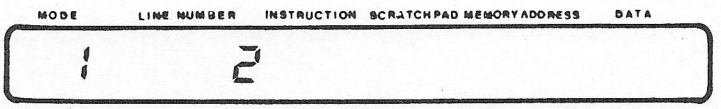
4.3.2 Changing program words

It is assumed that we want to change the hundreds up-counter of Example 4-1 in the previous section into a hundreds down-counter. This means that lines nos. 0002 and 0003 must be modified. The following example shows that the procedure is simple and straightforward.

Example 4-2 Changing program words.

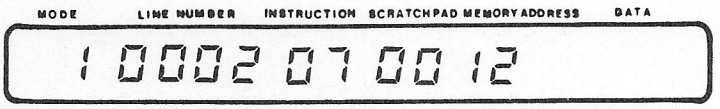
Select line no. 0002:

[LINE] [2]



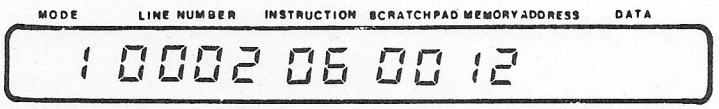
Display contents:

[ENTER]



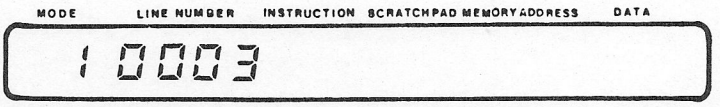
Type in new program word, check display, enter and recheck dispay:

[CNTD] [1] [2] [ENTER]



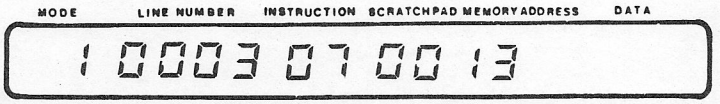
Select next line no.:

[S+]



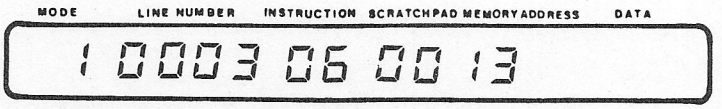
Display contents:

[ENTER]



Type in new program word, check display, enter and recheck display

[CNTD] [1] [3] [ENTER]



4.3.3 Deleting program words

It may be desirable in the test phase to remove program parts that do not perform as expected. With the PU20, program parts are easily deleted by using the DEL(ETE) key. The following steps are taken:

1. Enter first line number of the program section to be removed. The 'RUN' LED will light.
2. Press DEL key. The abbreviation 'DEL' will now appear in the DATA field.
3. Type in on the digit keyboard the number of program words to be deleted. Check this number, which will be displayed in the INST field.
4. Depress ENTER key. Now, the 'BUSY' LED will become illuminated. Meanwhile, the undesired program part will be deleted and the remainder of the program until the next NOP word shifted so that it will join the previous program section. NOP instructions will be created to fill the open line numbers. Deletion completed, both the 'RUN' and 'BUSY' LED will extinguish while the acronym 'EOJ' – for End Of Job – appears in the DATA field.

Fig. 4-5 illustrates what occurs in step 4, where it is assumed that program part B containing fifty lines, nos. 301 to 350, is to be removed. It is seen that program C is shifted to join program A, while new NOP words appear on lines 973 to 1022.

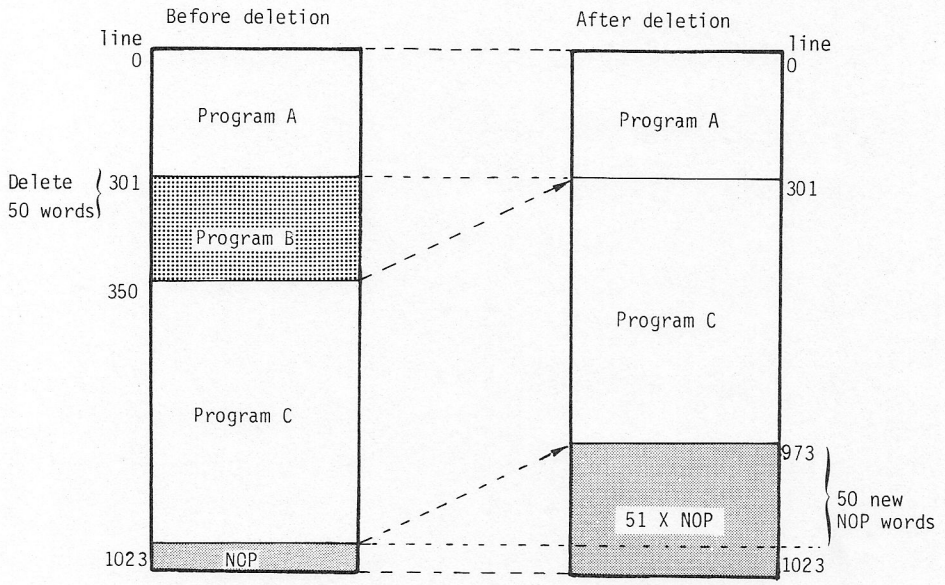


Fig. 4-5 Delete action of PU20 for program B (50 lines) to be removed, one NOP word being available at last line no. 1023 of the program.

It is clear from what has been discussed that the presence of a NOP instruction in the program is essential for correct delete action, or the entire program will become upset. ('BUSY' LED will stay on, in which case press RESET key to extinguish LED).

Therefore, a NOP instruction should be placed on the highest line number

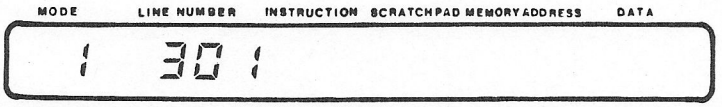
It should be noted that, with a large program memory and a NOP word placed on the highest line number, deletion will take a longer time than in the case of a 1k memory.

Example 4-3 helps to explain Fig. 4-5.

Example 4-3 Delete program words.

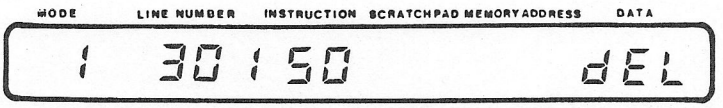
Select first line no. of program part to be deleted:

LINE 3 0 1



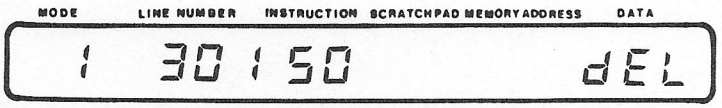
Press DEL key and enter number of words to be deleted:

DEL 5 0

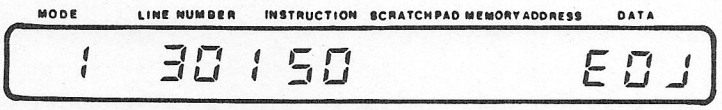


Start delete operation:

ENTER



Deletion ready ——————→



Deleting more than 99 program words. Up to 99 words of program can be deleted in one operation. If more words are to be removed, the operation must be repeated entering on the digit keyboard the number of remaining program words to disappear. Where the program section to be deleted contains NOP words, it is recommended for proper delete action to select for the second delete operation a line that is a number 99 higher than that taken for the first operation. Fig. 4-6 shows what occurs if two delete commands are successively given on the *same* line. Program part B' will remain because new NOP words are only generated on *lower* line numbers adjacent to the line originally containing a NOP word, *not* on higher line numbers.

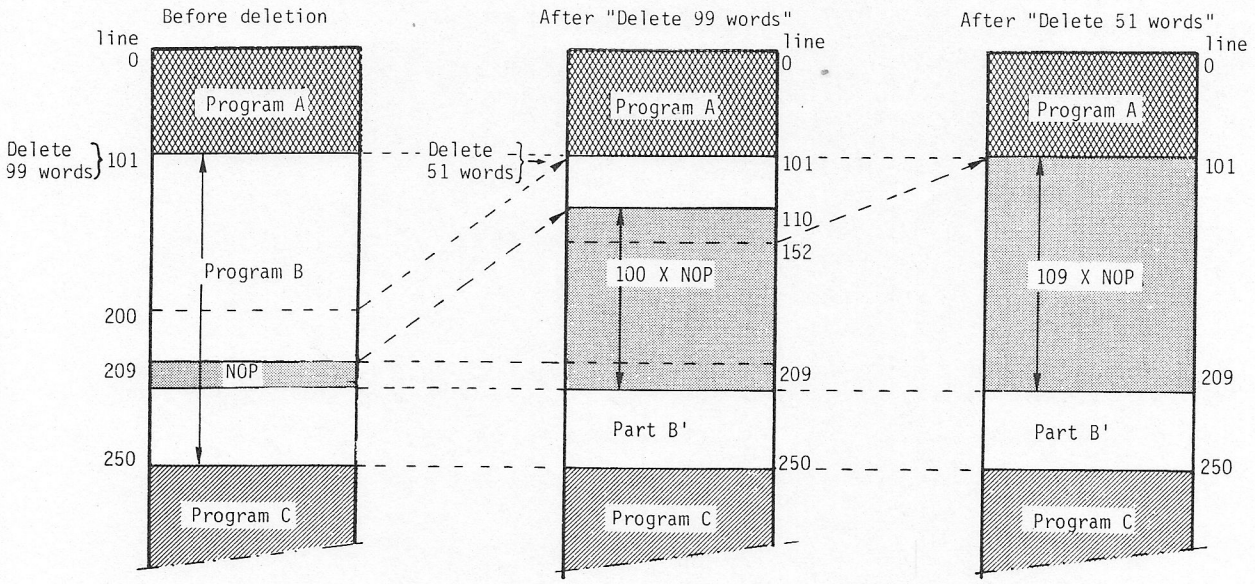


Fig. 4-6 Delete action of PU20 for program B to be removed (150 lines). First NOP word in that program is more than 99 lines away from line number where 'DELETE 99 words' command is given. Note that program part B' below section containing NOP words cannot be deleted (only 9 new NOP words generated in second delete operation).