μPD70108 (V20™)  
8/16-Bit CMOS Microprocessor

μPD70116 (V30™)  
16-Bit CMOS Microprocessor
μPD70108 (V20)
8/16-Bit CMOS Microprocessor

μPD70116 (V30)
16-Bit CMOS Microprocessor

October 1986
Stock No. 500350
©1986 NEC Electronics Inc./Printed in U.S.A.
CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction ........................................ 1-1</td>
</tr>
<tr>
<td>2</td>
<td>Pin Description ....................................... 2-1</td>
</tr>
<tr>
<td>3</td>
<td>Functional Description ........................................ 3-1</td>
</tr>
<tr>
<td></td>
<td>Execution Unit (EXU) ....................................... 3-1</td>
</tr>
<tr>
<td></td>
<td>Bus Control Unit (BCU) .................................... 3-4</td>
</tr>
<tr>
<td>4</td>
<td>Memory and I/O Configuration .......................... 4-1</td>
</tr>
<tr>
<td></td>
<td>Memory Configuration ..................................... 4-1</td>
</tr>
<tr>
<td></td>
<td>Memory Accessing ........................................ 4-2</td>
</tr>
<tr>
<td></td>
<td>I/O Configuration and Accessing ....................... 4-3</td>
</tr>
<tr>
<td>5</td>
<td>Read/Write Timing ..................................... 5-1</td>
</tr>
<tr>
<td>6</td>
<td>Interrupts ............................................. 6-1</td>
</tr>
<tr>
<td></td>
<td>Maskable Interrupts ...................................... 6-2</td>
</tr>
<tr>
<td></td>
<td>BRK Flag (Single-Step Interrupt) ...................... 6-3</td>
</tr>
<tr>
<td></td>
<td>Interrupt Disable Timing ................................ 6-4</td>
</tr>
<tr>
<td></td>
<td>Interrupts During Block Instructions ................ 6-4</td>
</tr>
<tr>
<td>7</td>
<td>Reset Operation ...................................... 7-1</td>
</tr>
<tr>
<td>8</td>
<td>Native and ( \mu )PD8080AF ............................... 8-1</td>
</tr>
<tr>
<td></td>
<td>Emulation Modes ......................................... 8-1</td>
</tr>
<tr>
<td></td>
<td>Native and 8080 Mode Shifting ......................... 8-1</td>
</tr>
<tr>
<td></td>
<td>Native to 8080 Emulation Mode ......................... 8-2</td>
</tr>
<tr>
<td>9</td>
<td>Standby Mode .......................................... 9-1</td>
</tr>
<tr>
<td></td>
<td>Entering Standby Mode .................................. 9-1</td>
</tr>
<tr>
<td></td>
<td>Status Signals in Standby Mode ....................... 9-1</td>
</tr>
<tr>
<td></td>
<td>Exiting Standby Mode by External Interrupts ......... 9-1</td>
</tr>
<tr>
<td></td>
<td>Exiting Standby Mode by Reset ........................ 9-1</td>
</tr>
<tr>
<td>10</td>
<td>Logical and Physical Addresses ...................... 10-1</td>
</tr>
<tr>
<td></td>
<td>Physical Address Generation .......................... 10-1</td>
</tr>
<tr>
<td></td>
<td>Memory Segments ........................................ 10-1</td>
</tr>
<tr>
<td>11</td>
<td>Addressing Modes ..................................... 11-1</td>
</tr>
<tr>
<td></td>
<td>Instruction Address .................................... 11-1</td>
</tr>
<tr>
<td></td>
<td>Memory Operand Address .............................. 11-2</td>
</tr>
<tr>
<td>12</td>
<td>Instruction Set ........................................ 12-1</td>
</tr>
<tr>
<td></td>
<td>Data Transfer ........................................... 12-4</td>
</tr>
<tr>
<td></td>
<td>Repeat Prefixes ......................................... 12-14</td>
</tr>
<tr>
<td></td>
<td>Primitive Block Transfer ................................ 12-16</td>
</tr>
<tr>
<td></td>
<td>Bit Field Manipulation ................................ 12-21</td>
</tr>
<tr>
<td></td>
<td>Input/Output ........................................... 12-25</td>
</tr>
<tr>
<td></td>
<td>Primitive Input/Output ................................ 12-28</td>
</tr>
<tr>
<td></td>
<td>Addition/Subtraction .................................. 12-29</td>
</tr>
<tr>
<td></td>
<td>BCD Arithmetic .......................................... 12-42</td>
</tr>
<tr>
<td></td>
<td>Increment/Decrement ................................... 12-47</td>
</tr>
<tr>
<td></td>
<td>Multiplication .......................................... 12-50</td>
</tr>
<tr>
<td></td>
<td>Division ................................................ 12-56</td>
</tr>
<tr>
<td></td>
<td>BCD Adjust ............................................... 12-61</td>
</tr>
<tr>
<td></td>
<td>Data Conversion ........................................ 12-63</td>
</tr>
<tr>
<td></td>
<td>Comparison ............................................... 12-65</td>
</tr>
<tr>
<td></td>
<td>Complement Operation .................................. 12-68</td>
</tr>
<tr>
<td></td>
<td>Logical Operation ...................................... 12-70</td>
</tr>
<tr>
<td></td>
<td>Bit Manipulation ....................................... 12-81</td>
</tr>
<tr>
<td></td>
<td>Shift ................................................. 12-100</td>
</tr>
<tr>
<td></td>
<td>Rotate ................................................ 12-118</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Instruction Set (cont) ................................ 12-162</td>
</tr>
<tr>
<td></td>
<td>Subroutine Control ..................................... 12-142</td>
</tr>
<tr>
<td></td>
<td>Stack Operation ........................................ 12-146</td>
</tr>
<tr>
<td></td>
<td>Branch .................................................. 12-154</td>
</tr>
<tr>
<td></td>
<td>Conditional Branch .................................... 12-157</td>
</tr>
<tr>
<td></td>
<td>Break .................................................. 12-167</td>
</tr>
<tr>
<td></td>
<td>CPU Control ............................................. 12-171</td>
</tr>
<tr>
<td></td>
<td>Segment Override Prefix ............................... 12-176</td>
</tr>
<tr>
<td></td>
<td>Emulation Mode ........................................ 12-176</td>
</tr>
</tbody>
</table>

Appendix

A \( \mu \)PD70108/70116 Instruction Index ........ A-1

Table

1-1 \( \mu \)PD70108/70116 Pin Identification ............. 1-2
4-1 Data Type and Addressing .................................. 4-1
6-1 Interrupt Sources ......................................... 6-1
9-1 Signal Status in Standby Mode ............................ 9-1
12-1 Operand Types ........................................... 12-1
12-2 Instruction Words ....................................... 12-2
12-3 Operation Description .................................. 12-2
12-4 Flag Operations ......................................... 12-3
12-5 Memory Addressing ...................................... 12-3
12-6 Selection of 8- and 16-Bit Registers ................. 12-3
12-7 Selection of 8-Bit and Segment Register ............. 12-3

Figure

1-1 \( \mu \)PD70108/70116 Simplified Block Diagram .......... 1-1
1-2 Pin Configuration, 40-Pin Plastic or Ceramic DIP ...... 1-2
1-3 Pin Configuration, 44-Pin PLCC .......................... 1-3
1-4 Pin Configuration, 52-Pin Plastic Mini flat ............ 1-3
3-1 \( \mu \)PD70108/70116 Block Diagram ........................ 3-2
3-2 Effective Address Generator ............................. 3-3
4-1 Memory Map ............................................. 4-1
4-2 Word and Double Word Placement in Memory ............. 4-1
4-3 \( \mu \)PD70108 Memory Interface ............................ 4-2
4-4 \( \mu \)PD70116 Memory Interface ............................ 4-2
4-5 I/O Map .............................................. 4-3
5-1 Read Timing of \( \mu \)PD70108 Memory and I/O (Small-Scale Systems) .......... 5-1
5-2 Write Timing of \( \mu \)PD70108 Memory and I/O (Small-Scale Systems) ...... 5-1
5-3 Read Timing of \( \mu \)PD70116 Memory and I/O (Small-Scale Systems) ...... 5-1
5-4 Write Timing of \( \mu \)PD70116 Memory and I/O (Small-Scale Systems) ...... 5-1
**CONTENTS (cont)**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-5</td>
<td>5-2</td>
<td>8-1</td>
<td>8-1</td>
</tr>
<tr>
<td>5-6</td>
<td>5-2</td>
<td>8-2</td>
<td>8-1</td>
</tr>
<tr>
<td>5-7</td>
<td>5-2</td>
<td>8-3</td>
<td>8-2</td>
</tr>
<tr>
<td>5-8</td>
<td>5-2</td>
<td>8-4</td>
<td></td>
</tr>
<tr>
<td>6-1</td>
<td>6-1</td>
<td>8-5</td>
<td></td>
</tr>
<tr>
<td>6-2</td>
<td>6-2</td>
<td>8-6</td>
<td></td>
</tr>
<tr>
<td>6-3</td>
<td>6-2</td>
<td>8-6</td>
<td></td>
</tr>
</tbody>
</table>

**Revision History**

Aug 1985  Original Issue

Sep 1986  Table 1-1 revised, figures 1-3 and 1-4 added, Section 2 revised and re-arranged. Marginal arrows (↑) identify significant changes in Sections 1, 2, and 12.
Description

The µPD70108 (V20) and µPD70116 (V30) are high-performance, low-power CMOS microprocessors with a 16-bit internal architecture. The µPD70108 has an 8-bit external data bus and the µPD70116 has a 16-bit external data bus. Figure 1 is a simplified block diagram.

The µPD70108/70116 has a powerful instruction set that is a superset of the µPD8086/8088 instruction set and provides the following enhanced operations:

- Multidigit BCD addition, subtraction, comparison of 1- to 254-digit BCD strings
- High-speed multiplication/division
- Bit field manipulations
  - Data transfer of 1- to 16-bit fields between memory and accumulator
- Bit manipulation instructions
  - 8- or 16-bit register/memory operands
  - Set, clear, invert, or test any bit

Dedicated hardware performs high-speed multiplication/division (4 to 6 µs at 8 MHz) and effective address calculation. In addition, an internal dual bus system reduces processing time.

The µPD70108/70116 has three operating modes: native, emulation, and standby. Native mode executes the µPD70108/70116 instruction set; emulation mode directly executes the µPD8080AF instruction set. The standby mode significantly reduces power consumption.

Features

- 101 instructions
- 250-ns instruction execution time (8-MHz clock)
- 1-Mbyte addressable memory
- Various memory addressing modes
- 14- x 16-bit register set
- High-speed block transfers
  - µPD70108: 1.0 Mbytes/second (at 8 MHz)
  - µPD70116: 2.0 Mbytes/second (at 8 MHz)
- Various interrupt processing functions
- IEE-796 bus-compatible interface
- 5-, 8-, 10-MHz clock
- 40-pin plastic/ceramic DIP, 44-pin PLCC, and 52-pin plastic miniflat packages
- Single +5-volt power source

V20 and V30 are trademarks of NEC Corporation.

Figure 1-1. µPD70108/70116 Simplified Block Diagram

Pin Identification

Table 1 lists pins in alphabetical order by symbol and briefly describes pin functions. Section 2 gives additional descriptions.

Figures 1-2, 1-3, and 1-4 are pin configuration drawings of the four package types: 40-pin plastic or ceramic DIP, 44-pin plastic leaded chip carrier (PLCC), and 52-pin plastic miniflat.
**Table 1-1. μPD70108/70116 Pin Identification**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Direction</th>
<th>Function (Note 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_{19..A_{16}/PS_3..PS_0}</td>
<td>Out</td>
<td>High-order address bits/Processor status bits</td>
</tr>
<tr>
<td>A_{15..A_8} (Note 1)</td>
<td>Out</td>
<td>Middle address bits</td>
</tr>
<tr>
<td>AD_{7..AD_0} (Note 1)</td>
<td>In/Out</td>
<td>Address/data bus</td>
</tr>
<tr>
<td>AD_{15..AD_0} (Note 2)</td>
<td>In/Out</td>
<td>Address/data bus</td>
</tr>
<tr>
<td>ASTB (QS_0)</td>
<td>Out</td>
<td>Address strobe (Queue status bit 0)</td>
</tr>
<tr>
<td>BUFEN (BS_0)</td>
<td>Out</td>
<td>Buffer enable (Bus status bit 0)</td>
</tr>
<tr>
<td>BUF\textsuperscript{R}/W (BS_1)</td>
<td>Out</td>
<td>Buffer read/write (Bus status bit 1)</td>
</tr>
<tr>
<td>CLK</td>
<td>In</td>
<td>Clock</td>
</tr>
<tr>
<td>GND</td>
<td>Ground</td>
<td></td>
</tr>
<tr>
<td>HLDAK (RQ/AK_1)</td>
<td>Out (In/Out)</td>
<td>Hold acknowledge output (Bus hold request input/Acknowledge output 1)</td>
</tr>
<tr>
<td>HLDRO (RQ/AK_0)</td>
<td>In (In/Out)</td>
<td>Hold request input (Bus hold request input/Acknowledge output 0)</td>
</tr>
<tr>
<td>IC</td>
<td>Internally connected (Note 4)</td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td>In</td>
<td>Maskable interrupt</td>
</tr>
<tr>
<td>INTAK (QS_1)</td>
<td>Out</td>
<td>Interrupt acknowledge (Queue status bit 1)</td>
</tr>
<tr>
<td>IO/\textsuperscript{M} (BS_2) (Note 1)</td>
<td>Out</td>
<td>Access is I/O or memory (Bus status bit 2)</td>
</tr>
<tr>
<td>IO/\textsuperscript{M} (BS_2) (Note 2)</td>
<td>Out</td>
<td>Access is I/O or memory (Bus status bit 2)</td>
</tr>
<tr>
<td>LBSO (HIGH) (Note 1)</td>
<td>In</td>
<td>Latched bus status 0 (Always high)</td>
</tr>
<tr>
<td>NC</td>
<td>Not connected</td>
<td></td>
</tr>
<tr>
<td>NMI</td>
<td>In</td>
<td>Nonmaskable interrupt</td>
</tr>
<tr>
<td>POLL</td>
<td>In</td>
<td>Poll</td>
</tr>
<tr>
<td>RD</td>
<td>Out</td>
<td>Read strobe</td>
</tr>
<tr>
<td>READY</td>
<td>In</td>
<td>Ready</td>
</tr>
<tr>
<td>RESET</td>
<td>In</td>
<td>Reset</td>
</tr>
<tr>
<td>S/LG</td>
<td>In</td>
<td>Small-scale system input/Large-scale system input</td>
</tr>
<tr>
<td>UBE (Note 2)</td>
<td>In</td>
<td>Upper byte enable</td>
</tr>
<tr>
<td>V_{DD}</td>
<td>+5-volt power supply</td>
<td></td>
</tr>
<tr>
<td>WR (BUSLOCK)</td>
<td>Out</td>
<td>Write strobe (Bus lock)</td>
</tr>
</tbody>
</table>

**Note:**

1. μPD70108 only.
2. μPD70116 only.
3. Where pins have different functions in small- and large-scale systems, the large-scale system pin symbol and function are in parentheses.

**Figure 1-2. Pin Configuration, 40-Pin Plastic or Ceramic DIP**

```plaintext

<table>
<thead>
<tr>
<th>IC</th>
<th>1</th>
<th>40</th>
<th>V_{DD}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_{14}</td>
<td>2</td>
<td>39</td>
<td>A_{15}</td>
</tr>
<tr>
<td>A_{13}</td>
<td>3</td>
<td>38</td>
<td>A_{16}/PS_0</td>
</tr>
<tr>
<td>A_{12}</td>
<td>4</td>
<td>37</td>
<td>A_{17}/PS_1</td>
</tr>
<tr>
<td>A_{11}</td>
<td>5</td>
<td>36</td>
<td>A_{18}/PS_2</td>
</tr>
<tr>
<td>A_{10}</td>
<td>6</td>
<td>35</td>
<td>A_{19}/PS_3</td>
</tr>
<tr>
<td>A_9</td>
<td>7</td>
<td>34</td>
<td>LBSO (HIGH)</td>
</tr>
<tr>
<td>A_8</td>
<td>8</td>
<td>33</td>
<td>S/LG</td>
</tr>
<tr>
<td>A_{7}</td>
<td>9</td>
<td>32</td>
<td>RD</td>
</tr>
<tr>
<td>A_{6}</td>
<td>10</td>
<td>31</td>
<td>HLDRO (RQ/AK_0)</td>
</tr>
<tr>
<td>A_{5}</td>
<td>11</td>
<td>30</td>
<td>HLDAK (RQ/AK_1)</td>
</tr>
<tr>
<td>A_{4}</td>
<td>12</td>
<td>29</td>
<td>WR (BUSLOCK)</td>
</tr>
<tr>
<td>A_{3}</td>
<td>13</td>
<td>28</td>
<td>IO/\textsuperscript{M} (BS_2)</td>
</tr>
<tr>
<td>A_{2}</td>
<td>14</td>
<td>27</td>
<td>BUF\textsuperscript{R}/W (BS_1)</td>
</tr>
<tr>
<td>A_{1}</td>
<td>15</td>
<td>26</td>
<td>BUFEN (BS_0)</td>
</tr>
<tr>
<td>NMI</td>
<td>16</td>
<td>25</td>
<td>ASTB (QS_0)</td>
</tr>
<tr>
<td>\textsuperscript{INT}</td>
<td>17</td>
<td>24</td>
<td>INTAK (QS_1)</td>
</tr>
<tr>
<td>\textsuperscript{INT}</td>
<td>18</td>
<td>23</td>
<td>POLL</td>
</tr>
<tr>
<td>CLK</td>
<td>19</td>
<td>22</td>
<td>READY</td>
</tr>
<tr>
<td>GND</td>
<td>20</td>
<td>21</td>
<td>RESET</td>
</tr>
</tbody>
</table>

(4) IC should be connected to ground.
(5) Unused input pins should be tied to ground or V_{DD} to minimize power dissipation and prevent potentially harmful current flow.
```
**Section 1**

**Introduction**

- **Figure 1-3.** Pin Configuration, 44-Pin Plastic Leaded Chip Carrier (PLCC)
- **Figure 1-4.** Pin Configuration, 52-Pin Plastic Miniflat
This section describes the functions of input and output signals. Descriptions are in alphabetical order by pin symbol. Unless otherwise specified, they apply to μPD70108 and μPD70116 in small-scale and large-scale systems.

The width of the data bus is different for the μPD70108 and μPD70116. Therefore, each microprocessor uses the address/data bus in a different manner.

Memory identification signals for the two processors are also different. The μPD70108 uses an 10/M signal; the μPD70116 uses an I0/M signal.

**A19-A16/PS3-PS0 [Address Bus/Processor Status]**

These lines are time-multiplexed to operate as an address bus and also to output the processor status signals.

When used as the address bus, A19-A16 are the four high-order bits of the 20-bit memory address. During an I/O bus cycle all four bits are 0.

Processor status signals are for memory and I/O use. PS3 is always 0 in the native mode and always 1 in the emulation mode. The contents of the interrupt enable flag (IE) are carried via PS2. Signals PS1 and PS0 indicate which memory segment is being accessed.

<table>
<thead>
<tr>
<th>A17-PS1</th>
<th>A16-PS0</th>
<th>Memory Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Data segment 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Stack segment</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Program segment</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Data segment 0</td>
</tr>
</tbody>
</table>

These pins are tri-state and become high impedance during hold acknowledge.

**A15-A8 [Address Bus]**

In the μPD70108 only, A15-A8 are the middle 8 bits of the 20-bit memory address. This bus is tri-state and becomes high impedance during hold acknowledge. An address bit is 1 when high and 0 when low.

**AD7-AD0 [Address/Data Bus]**

In the μPD70108 only, AD7-AD0 is a time-multiplexed address/data bus. These lines output either the lower 8 bits of the 20-bit address or 8 bits of data. Input/output of 16-bit data is performed in two steps: low byte followed by high byte.

This is a tri-state bus and becomes high impedance during hold and interrupt acknowledge. An AD bit is 1 when high and 0 when low.

**AD15-AD0 [Address/Data Bus]**

In the μPD70116 only, AD15-AD0 is a time-multiplexed address/data bus. An AD bit is 1 when high and 0 when low. The bus contains the lower 16 bits of the 20-bit address during T1 of the bus cycle. The bus is used as a 16-bit data bus during T2, T3, and T4 of the bus cycle. The address/data bus is tri-state and can be at a high or low level in standby mode. The bus is at high impedance during hold acknowledge and interrupt acknowledge.

**ASTB [Address Strobe]**

In a small-scale system, the CPU generates ASTB to latch address information at an external latch. ASTB is held to a low level in standby mode.

**BS2-BS0 [Bus Status]**

In a large-scale system, the CPU uses these status signals to allow an external bus controller to monitor the current bus cycle. The external bus controller decodes BS2-BS0 and generates the control signals required to perform a memory or I/O device access.

The BS2-BS0 signals are tri-state outputs and become high impedance during hold acknowledge. They are held to a high level in the standby mode.

<table>
<thead>
<tr>
<th>BS2</th>
<th>BS1</th>
<th>BS0</th>
<th>Bus Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Interrupt acknowledge</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>I/O read</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>I/O write</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Halt</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Program fetch</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Memory read</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Memory write</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Passive state</td>
</tr>
</tbody>
</table>

**BUFEN [Buffer Enable]**

In a small-scale system, BUFEN is used as the output enable signal for an external bidirectional buffer. The CPU generates this signal during data transfer operations with an external memory or I/O device or during the input of an interrupt vector.

The BUFEN signal is held to a high level in the standby mode and becomes high impedance during hold acknowledge.
---

**μPD70108/70116**

**BUFFR/W [Buffer Read/Write]**

In a small-scale system, the level of this signal determines the direction of data transfer with an external bidirectional buffer. A high signal specifies data transmission from the CPU to an external device. A low signal specifies data transmission from the external device to the CPU.

This output can be a high or low level in the standby mode. It becomes high impedance during hold acknowledge.

**BUSLOCK [Bus Lock]**

In a large-scale system, the CPU uses this signal to secure the bus while executing the instruction immediately following the BUSLOCK prefix. The signal inhibits other bus masters in a multiprocessor system from using the system bus during this time. The output is held to a high level in the standby mode, but is a low level if the BUSLOCK instruction is executed immediately before a HALT instruction.

The signal is tri-state and becomes high impedance during hold acknowledge.

**CLK [Clock]**

The CLK pin is the external clock input.

**HLDACK [Hold Acknowledge]**

In a small-scale system, HLDACK indicates the CPU has accepted a hold request signal (HLDREQ). While HLDACK is high, the address bus, address/data bus, and control lines are held in the high-impedance state.

**HLDREQ [Hold Request]**

In a small-scale system, external devices input the HLDREQ signal to request that the CPU release the address, address/data, and control buses.

**IC [Internally Connected]**

The IC pin is used for factory tests. Normally, the μPD70108/70116 is used with this pin at ground potential.

**INT [Maskable Interrupt]**

The INT pin is used for interrupt requests that can be masked by software. This input is an active high level and is sensed during the last clock of the current instruction. The interrupt will be accepted if the system is in the interrupt enable state (interrupt enable flag IE = 1). The CPU generates INTAK to notify external devices that the interrupt request is being acknowledged. INT must be held high until the INTAK signal is returned.

If NMI and INT interrupts occur at the same time, NMI has priority and INT will not be accepted. A hold request will be accepted even during interrupt acknowledge.

INT causes the microprocessor to exit the standby mode.

**INTAK [Interrupt Acknowledge]**

In a small-scale system, when the CPU accepts an INT signal, it asserts the INTAK signal active low. The interrupting device synchronizes with the signal and puts the interrupt vector number on the data bus (AD7-AD0).

During standby mode, INTAK is held to a high level.

**IO/M [IO/Memory]**

In a small-scale system, μPD70108 system, the CPU outputs this signal to indicate either an I/O or memory access. A high-level output specifies an I/O access and a low-level output specifies a memory access. This output can be a high or low level in the standby mode.

The pin is tri-state and becomes high impedance during hold acknowledge.

**IO/ M [IO/Memory]**

In a small-scale μPD70116 system, the CPU generates this signal to specify either an I/O access or a memory access. A low-level output specifies an I/O access and a high-level output specifies a memory access. The output can be a high or low level in the standby mode.

The pin is tri-state and becomes high impedance during hold acknowledge.
**LBSO [Latched Bus Status 0]**

In a small-scale \( \mu \)PD70108 system, the CPU uses this signal (along with the IO/M and BUFR/W signals) to inform external devices of the status of the current bus cycle. See below.

<table>
<thead>
<tr>
<th>IO/M</th>
<th>BUFR/W</th>
<th>LBSO</th>
<th>BUS Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Program fetch</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Memory read</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Memory write</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Passive state</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Interrupt acknowledge</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>I/O read</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>I/O write</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Held</td>
</tr>
</tbody>
</table>

**NMI [Nonmaskable Interrupt]**

The NMI signal is used for interrupt requests that cannot be masked by software. The interrupt is triggered on the rising edge of NMI and can be sensed during any clock cycle. NMI must be held high for at least five clock cycles after its rising edge. Actual interrupt processing begins after completion of the instruction in progress.

The contents of interrupt vector 2 determines the starting address for the interrupt servicing routine. A hold request will be accepted even during NMI acknowledge. This interrupt will cause the microprocessor to exit the standby mode.

**POLL [Poll]**

The CPU checks the input at this pin when executing the POLL instruction. If the input is low, execution continues. If the input is high, the CPU will check the state of the input every five clock cycles until the input again becomes low.

These functions synchronize CPU program execution with the operation of external devices.

**QS1, QS0 [Queue Status]**

In a large-scale system, the CPU uses QS1 and QS0 to allow external devices, such as the floating-point arithmetic processor chip, to monitor the status of the internal CPU instruction queue.

<table>
<thead>
<tr>
<th>QS1</th>
<th>QS0</th>
<th>Instruction Queue Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>NOP (queue did not change)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>First byte of an instruction taken from queue</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Flush queue</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Subsequent byte of instruction taken from queue</td>
</tr>
</tbody>
</table>

The instruction queue status indicated by these signals is the status when the execution unit (EXU) accesses the instruction queue. The data output from QS0 and QS1 is therefore valid only for one clock immediately following queue access.

QS1 and QS0 enable the floating-point processor chip to monitor the CPU's program execution status. In this manner, the floating-point processor can synchronize its operation with the CPU whenever it gains control from a floating-point operation instruction (FPO).

QS1 and QS0 are held to a low level during standby mode.

**RD [Read Strobe]**

The CPU outputs the RD signal during a data read from an I/O device or memory. The IO/M or I/M signal determines whether the read is I/O or memory. RD is a tri-state output and becomes high impedance during a hold acknowledge.

**READY [Ready]**

READY indicates that the data transfer is complete. A high indicates READY is true; a low indicates READY is false (not ready).

When READY goes high during a read cycle, the data is latched one clock cycle later and the bus cycle is terminated. When READY goes high during a write cycle, the bus cycle is terminated one clock cycle later.
**RESET [Reset]**

RESET is the CPU reset signal and is an active high level. This signal has priority over all other operations. After RESET returns to the low level, the CPU begins execution of the program starting at address FFF0H.

RESET causes the microprocessor to exit the standby mode.

**RQ/AK<sub>1</sub>, RQ/AK<sub>0</sub> Hold Request Acknowledge**

In a large-scale system, these pins function as the bus hold request inputs (RQ), and the bus hold acknowledge outputs (AK). The RQ/AK<sub>0</sub> signal has priority over the RQ/AK<sub>1</sub> signal.

These signals have tri-state outputs with on-chip pull-up resistors that keep the pins at a high level when the output is at the high-impedance state.

**S/LG [Small/Large]**

This signal determines the operating mode of the CPU. The signal is fixed at either a high or low level. When the signal is high level, the CPU operates in the small-scale system mode. When the signal is low level, the CPU operates in the large-scale system mode. A small-scale system will have at most one additional bus master requesting use of the bus. A large-scale system can have more than one.

As noted in table 1-1, some pins have different symbols and functions in small-scale and large-scale systems.

---

**UBE [Upper Byte Enable]**

UBE indicates the use of the upper 8 bits (AD<sub>15</sub>-AD<sub>8</sub>) of the data bus. This signal is active low during T1-T4 of the bus cycle. Bus cycles in which the signal is active are shown below:

<table>
<thead>
<tr>
<th>Type of Bus Operation</th>
<th>UBE</th>
<th>AD&lt;sub&gt;0&lt;/sub&gt;</th>
<th>Number of Bus Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word to even address</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Word to odd address</td>
<td>0*</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1**</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Byte to even address</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Byte to odd address</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

* First bus cycle  
** Second bus cycle

UBE goes low continuously during the interrupt acknowledge state. The signal is held high during standby mode. The signal is a tri-state output and becomes high impedance during a hold acknowledge.

Section 4, Memory Accessing, contains detailed information on the use of UBE.

**WR [Write Strobe]**

In a small-scale system, the CPU asserts WR during a write to an I/O device or memory. The IO/M or IO/M signal selects either I/O or memory. The WR output is held to a high level in the standby mode.

The pin is tri-state and becomes high impedance during hold acknowledge.
As shown in figure 3-1, the \( \mu PD70108 \) and \( \mu PD70116 \) both contain two internal, independent processing units: an execution unit (EXU) and a bus control unit (BCU).

The EXU controls the internal data processing that executes the instruction set of the \( \mu PD70108 \) and \( \mu PD70116 \).

The BCU is the interface between the EXU and the external bus. It prefetches instructions for the instruction queue — 4 bytes in the \( \mu PD70108 \) and 6 bytes in the \( \mu PD70116 \). It also accesses memory (upon request from the EXU) for additional operands, or stores EXU results.

**EXECUTION UNIT (EXU)**

The EXU includes the following functional elements:

- Program Counter
- General Purpose Registers (AW, BW, CW, DW)
- Pointers (SP, BP) and Index Registers (IX, IY)
- Temporary Register/Shifter (TA/TB)
- Temporary Register C (TC)
- Arithmetic and Logic Unit (ALU)
- Program Status Word (PSW)
- Loop Counter (LC)
- Effective Address Generator (EAG)
- Instruction Decoder
- Microaddress Register
- Microinstruction ROM
- Microinstruction Sequencer
- Dual data bus

**Program Counter (PC)**

The program counter is a 16-bit binary counter that contains the segment offset of the program memory address of the next instruction which the EXU is to execute.

The PC increments each time the microprogram fetches a byte from the instruction queue. A new location value is loaded into the PC each time a branch, call, return, or break instruction is executed. At this time, the contents of the PC are the same as the Prefetch Pointer (PFP).

**General Purpose Registers (AW, BW, CW, DW)**

There are four 16-bit general-purpose registers. Each one can be used as one 16-bit register or as two 8-bit registers. This is accomplished by dividing the registers into their high and low bytes (AH, AL, BH, BL, CH, CL, DH, DL).

Each register is also used as a default register for processing specific instructions. The default assignments are:

- **AH**  Byte multiplication/division
- **BW**  Translation
- **CW**  Loop control branch, repeat prefix
- **CL**  Shift instructions, rotation instructions, BCD operations
- **DW**  Word multiplication/division, indirect addressing, I/O

**Pointers (SP, BP) and Index Registers (IX, IY)**

These registers serve as base pointers or index registers when accessing memory using based, indexed, or base indexed addressing.

These registers can also be used for data transfer and arithmetic and logical operations in the same manner as the general-purpose registers. They cannot be used in these areas as 8-bit registers.

Also, the SP, IX, and IY registers act as default registers for specific operations. The default assignments are:

- **SP**  Stack operations
- **IX**  Block transfer, BCD string operations (source)
- **IY**  Block transfer, BCD string operations (destination)

**Temporary Register/Shifter (TA/TB)**

TA/TB are 16-bit temporary registers/shifters used in the execution of multiply/divide and shift/rotate (including BCD rotate) instructions. Execution of multiplication/division instructions can be accomplished approximately four times faster than when using the microprogramming method.

When executing a multiply or divide instruction, TA+TB operates as a 32-bit temporary register/shifter. TB operates as a 16-bit temporary register/shifter when executing shift/rotate instructions. Both TA and TB can be read from or written to. When this is done from the internal bus, the upper byte and lower byte may be accessed independently. The contents of TA and TB are inputs to the ALU.

**Temporary Register C (TC)**

The TC is a 16-bit temporary register used for internal processing such as a multiply or divide operation. The contents of TC are inputs to the ALU.

**Arithmetic and Logic Unit (ALU)**

The ALU consists of a full adder and a logical operation unit. The ALU performs the following arithmetic operations:
Figure 3-4. μPD70108/70116 Block Diagram

Notes:
1. μPD70108 only
2. μPD70116 only
Section 3
Functional Description

Add, subtract, multiply, and divide
Increment, decrement, and two's complement
The ALU also performs the following logical operations:
• AND, OR, XOR, complement
• Bit test, set, clear, and complement

Program Status Word (PSW)
The PSW contains six status flags:
• V (Overflow)
• S (Sign)
• Z (Zero)
• AC (Auxiliary carry)
• P (Parity)
• CY (Carry)
The program status word also contains four control flags:
• MD (Mode)
• DIR (Direction)
• IE (Interrupt enable)
• BRK (Break)
When the PSW is pushed onto the stack, the word format of the various flags is as follows:

<table>
<thead>
<tr>
<th>PSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>M D 1 1 V D I B S Z 0 A 0 P 1 C</td>
</tr>
<tr>
<td>D C R K</td>
</tr>
</tbody>
</table>

The status flags are set and reset depending on the result of each type of instruction executed. Instructions are provided that set, reset, and complement the CY flag directly. Other instructions set and reset the control flags and control the operation of the CPU.

Loop Counter (LC)
The loop counter (LC) is a 16-bit register which counts:
• Loop times specified in the primitive block transfer
• I/O instructions controlled with repeat prefix instructions such as REP and REPC
• Shifts for the multi-bit shift/rotate instructions
The processing speed for multiple-bit rotation of a register is approximately twice as fast as when using the microprogram method.

Example:
RORC AW, CL ;CL = 5

<table>
<thead>
<tr>
<th>Microprogram Method</th>
<th>Loop Counter Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 + (4 × 5) = 28 clocks</td>
<td>7 + 5 = 12 clocks</td>
</tr>
</tbody>
</table>

Effective Address Generator (EAG)
The effective address generator (EAG) performs a high-speed effective address calculation for memory access. While the microprogramming method normally requires 5 to 12 clock cycles to calculate an address, the EAG completes all the EA calculations in 2 clocks for all addressing modes (see figure 3-2).
The EXU fetches the instruction bytes that have the operand field and determines if the instruction will require a memory access. If it does, the EAG calculates the effective address and transfers it to the DP (data pointer) which generates control signals that handle the ALU and corresponding registers. In addition, if it is necessary, the EAG requests a bus cycle from the BCU.

Instruction Decoder
The instruction decoder decodes the first byte of an instruction into groups with specific functions and holds them during the instruction execution.

Microaddress Register
The microaddress register specifies the starting address in the microinstruction ROM of the next instruction to be executed. At the beginning of a new instruction, the first byte of the instruction is taken from the prefetch queue and put into the microaddress register. The register then specifies the starting address of the corresponding microinstruction sequence.

Microinstruction ROM
The microinstruction ROM has 1024 microinstructions. Each microinstruction is 29 bits wide.
Microinstruction Sequencer

The microinstruction sequencer controls the microaddress register operation, microinstruction ROM output, and the synchronization of the EXU with the BCU.

Dual Data Bus

The μPD70108/70116 contains a dual, 16-bit data bus that consists of a main and subdata bus. The dual data bus reduces the number of processing steps for instruction execution. For addition/subtraction and logical and comparison operations, processing time is approximately 30% faster then in single-bus systems.

Example:

```
ADD AW, BW ;AW ← AW + BW
```

<table>
<thead>
<tr>
<th>Single-Bus System</th>
<th>Dual-Bus System</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. TA ← AW</td>
<td>TA ← AW, TB ← BW</td>
</tr>
<tr>
<td>2. TB ← BW</td>
<td>AW ← TA + TB</td>
</tr>
<tr>
<td>3. AW ← TA + TB</td>
<td></td>
</tr>
</tbody>
</table>

BUS CONTROL UNIT (BCU)

The BCU includes the following functional elements:

- Prefetch Pointer (PFP)
- Prefetch Queue (Q0-Q3/Q0-Q5)
- Data Pointer (DP)
- Temporary Communication Register (TEMP)
- Segment Registers (PS, SS, DS0, DS1)
- Address Modifier (ADM)

Prefetch Pointer (PFP)

The PFP is a 16-bit binary counter that contains a program segment offset. The offset is used to calculate a physical address that the Bus Control Unit (BCU) uses to prefetch the next byte or word for the instruction queue. The contents of the PFP are an offset from the Program Segment register (PS).

The PFP is incremented each time the BCU prefetches an instruction from the program memory. A new location will be loaded into the PFP whenever a branch, call, return, or break instruction is executed — this provides a time savings of several clocks since the PC does not require adjustment. At that time, the contents of the PFP will be the same as those of the program counter (PC).

Prefetch Queue (Q0-Q3/Q0-Q5)

The μPD70108/70116 has a prefetch queue that can store 4/6 instruction bytes that are prefetched by the BCU. The instruction bytes stored in the queue are taken from the queue and executed by the EXU. The queue is cleared when a branch, call, return, or break instruction has been executed, or when an external interrupt has been acknowledged.

Normally, the μPD70108 prefetches a byte if the queue has one or more empty bytes. The μPD70116 prefetches if the queue has one or more empty words (two bytes). If the time required to prefetch the instruction code from external memory is less than the mean execution time of instructions executed sequentially, the actual instruction cycle will be shortened by the time needed to fetch the instructions. This occurs because the next instruction code to be executed by the EXU will be available in the queue immediately after the completion of the previous instruction. As a result, the processing speed is increased when compared with a conventional CPU where the fetch and execute times do not overlap.

The queuing effect will be lowered if there are many instructions which clear the queue; for example, a branch instruction, or a series of instructions with a short instruction time.

Data Pointer (DP)

The DP is a 16-bit register that contains the read/write addresses of variables. Effective addresses calculated by the effective address generator are transferred to the DP.

Temporary Communication Register (TEMP)

The TEMP is a 16-bit temporary register that stores data being transferred between the external data bus and the EXU.

The TEMP can be read from or written to independently by the upper or lower byte. Basically, the EXU completes a write operation by transferring data to the TEMP and completes a read operation by taking the data transferred to the TEMP from the external data bus.
Segment Registers (PS, SS, DS₀, DS₁)

The memory addresses accessed by the µPD70108/70116 are divided into 64 Kbyte logical segments. The starting (base) address of each segment is specified by a segment register. The offset from this starting address is specified by the contents of another register or by the effective address.

The µPD70108/70116 uses four types of segment registers:

<table>
<thead>
<tr>
<th>Segment Register</th>
<th>Default Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS (Program Segment)</td>
<td>PFP</td>
</tr>
<tr>
<td>SS (Stack Segment)</td>
<td>SP, Effective Address</td>
</tr>
<tr>
<td>DS₀ (Data Segment 0)</td>
<td>IX, Effective Address</td>
</tr>
<tr>
<td>DS₁ (Data Segment 1)</td>
<td>IY, Effective Address</td>
</tr>
</tbody>
</table>

Address Modifier (ADM)

The address modifier logic generates a physical memory or I/O address by adding the segment register and PFP (or DP) contents.
MEMORY CONFIGURATION

Memory contains instructions, interrupt handler start addresses, stack data, and general data. Some of this data is stored in bytes and other in words. The \( \mu P D70108/70116 \) can access up to 1 Mbyte (512 Kwords) of memory by using the 20-bit address bus (A\(_{19}\)–A\(_0\)).

As the memory map in figure 4-1 shows, the first 1 Kbytes of addresses (0H–3FFH) are used for the interrupt vector table. Parts of this area may also be used for other purposes in some systems. The 12 bytes from address FFFFF0H to FFFFFBH are always used by the CPU when it is reset, and therefore cannot be used for any other purpose.

The four bytes from addresses FFFFFCH to FFFFFFH are reserved for future use and are not available.

Memory data can be stored in both even (A\(_0\) = 0) and odd (A\(_0\) = 1) addresses. The area where the interrupt start addresses (interrupt vector table) are stored must use even addresses. The \( \mu P D70116 \) can access a word regardless of whether the word is at an even or odd address. This allows both even and odd addresses to be used for an instruction. Table 4-1 shows the type and configuration of data, and address requirements. Figure 4-2 shows the placement of word and double word data in memory.

Table 4-1 Data Type and Addressing

<table>
<thead>
<tr>
<th>Data</th>
<th>Address</th>
<th>Data Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Code</td>
<td>Even or odd</td>
<td>1-6 bytes</td>
</tr>
<tr>
<td>Interrupt Vector Table</td>
<td>Even</td>
<td>2 words/vector</td>
</tr>
<tr>
<td>Stack</td>
<td>Even or odd</td>
<td>Word</td>
</tr>
<tr>
<td>General Variable</td>
<td>Even or odd</td>
<td>Byte, word, or double word</td>
</tr>
</tbody>
</table>

Figure 4-2. Word and Double Word Placement in Memory

![Memory Map Diagram](image-url)
MEMORY ACCESSING

Since the μPD70108 data bus is only 8 bits wide, only one byte (8 bits) is accessed during one bus cycle. Two bus cycles are required to access a data word from either an even or odd address. Figure 4-3 shows the interface between memory and the μPD70108. Figure 4-4 shows the interface between memory and the μPD70116.

The address space for the μPD70116 is 1 Mbyte, but because the μPD70116 can transfer both bytes and words of data, the physical memory appears to be two banks, each containing 512 Kbytes of data (figure 4-4). Data lines D7-D0 are connected to the low-order memory bank and address bit A0 selects this bank when A0=0. Data lines D15-D8 are connected to the high-order memory bank and signal UBE is used to select this bank when UBE is low. Address bits 19-1 contain the physical address within a data bank where the byte of data is to be accessed.

**Figure 4-3. μPD70108 Memory Interface**

![μPD70108 Memory Interface Diagram](image)

**Figure 4-4. μPD70116 Memory Interface**

![μPD70116 Memory Interface Diagram](image)
The following chart shows how Ao and UBE are used. Memory transfer operations are described after the chart.

<table>
<thead>
<tr>
<th>Type of Bus Operation</th>
<th>UBE</th>
<th>Ao</th>
<th>Number of Bus Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word to even address</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Word to odd address</td>
<td>0*</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1**</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Byte to even address</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Byte to odd address</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Notes:  * First bus cycle  ** Second bus cycle

When transferring a word of data to an even address, the μPD70116 puts the low-order data byte on D7-D0, the high-order data byte on D15-D8, and sets both UBE and Ao to 0. In this manner, both the low- and high-order memory banks are simultaneously selected and the transfer is performed in one bus cycle.

The transfer of a word of data to an odd address requires two bus cycles. In the first cycle, the μPD70116 puts the low-order data byte on D15-D8, sets UBE to 0, sets Ao to one, and transfers the first byte to the high-order memory bank. In the second cycle, the μPD70116 increments the address by +1, puts the high-order data byte on D7-D0, sets UBE to 1, sets Ao to 0, and transfers the second byte to the low-order data bank.

When transferring a byte of data to an even address, the μPD70116 puts the data byte on D7-D0, sets UBE to 1, Ao to 0, and transfers the data byte to the low-order memory bank.

When transferring a byte of data to an odd address, the μPD70116 puts the data byte on D15-D8, sets UBE to 0, sets Ao to 1, and transfers the data byte to the high-order memory bank.

The μPD70116 normally prefetches instruction codes in words. However, if a branch operation to an odd address takes place, only one byte is fetched from that odd address. After that, instruction codes are prefetched in words.

When the interrupt vector table is accessed in response to an interrupt, even addresses are always used. During an interrupt, two bus cycles are required because two words (segment base, and offset) are required.

One memory bus cycle requires four clocks. Thus, each time a word from an odd address is accessed, four additional clocks are required than when accessing an even-address word. When transferring a word from one memory area to another, the memory must be accessed twice. The word must be read from the source first and then written to the destination. If both the source and the destination are odd addresses, the execution time will be maximized. The following example shows the number of clocks required to execute the MOV reg, mem instruction for both a byte and word of data.

<table>
<thead>
<tr>
<th>Data</th>
<th>Processor</th>
<th>Number of Clocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes</td>
<td>μPD70108/70116</td>
<td>11</td>
</tr>
<tr>
<td>Words</td>
<td>μPD70116 (even address)</td>
<td>11</td>
</tr>
<tr>
<td>Words</td>
<td>μPD70108/70116 (odd address)</td>
<td>15</td>
</tr>
</tbody>
</table>

The above stack information is also true during a stack operation since all stack data is organized as words. Twice as many bus cycles are required during a stack operation using an odd rather than even address.

I/O CONFIGURATION AND ACCESSING

The μPD70108/70116 can access up to 64 Kbytes (32 Kwords) of I/O address area independent of memory. However, the upper 256 bytes (FF00H-FFFFH) are reserved by NEC for future use. The I/O address area is addressed by the lower 16 bits of the address bus. Figure 4-5 shows the I/O map.

Unlike memory, segment registers are not used in I/O. When the address bus carries I/O addresses, address bits A19-A16 are all zeros. Since data is transferred between the CPU and I/O in bytes or words, both 8-bit and 16-bit I/O devices can be connected to the μPD70116. Only 8-bit I/O devices can be connected to the μPD70108.

In the μPD70116, only one bus cycle is required to access a word on an even address; two bus cycles are required to access a word on an odd address.

When the μPD70116 accesses an 8-bit I/O device, bits Ao and UBE select the device. Bit A1 and higher bits select a device and the registers within that device. When
accessing 8-bit I/O devices, only even addresses should be assigned to the device and its internal registers. This allows the registers to be selected using only even addresses. Similarly, 8-bit I/O devices with internal registers assigned odd addresses must be accessed using odd addresses.

If a memory-mapped I/O configuration (memory address space allocated to an I/O device) is used, the I/O addresses can be allocated to a portion of the 1 Mbyte memory area. In this manner, all CPU addressing modes and instructions can be directly performed on the I/O device. For example, if a bit operation instruction for memory is used, one line of a specific I/O port can be tested for 1 or 0, set to 1, cleared to 0, or inverted. In a memory-mapped I/O configuration, control signals from the CPU are used exactly as for memory. Therefore, the I/O device is distinguished from memory only by its address. Care must be taken so that addresses of variables or the stack do not conflict with the addresses allocated to a memory-mapped I/O device.
Bus Cycles and Memory Access

One bus cycle is required for each access (read/write) of memory or I/O. A bus cycle is basically made up of four states (clocks): T1 through T4. When the microprocessor operates at 8 MHz, one state is 125 ns. The µPD70108 and µPD70116 fetch instructions and read data, using exactly the same timing (figures 5-1, 5-3, 5-5, and 5-7).

The EXU fetches an instruction from the instruction queue and executes it. The BCU continues prefetching instructions for the instruction queue until the queue becomes full. If the EXU does not fetch an instruction from the queue because another instruction is still being executed and the instruction queue is full, the BCU will not prefetch the next instruction. Instead, it automatically inserts an idle state (T1) after state T3. More idle states

---

**Figure 5-1. Read Timing of µPD70108 Memory and I/O (Small-Scale Systems)**

---

**Figure 5-2. Write Timing of µPD70108 Memory and I/O (Small-Scale Systems)**

---

**Figure 5-3. Read Timing of µPD70116 Memory and I/O (Small-Scale Systems)**

---

**Figure 5-4. Write Timing of µPD70116 Memory and I/O (Small-Scale Systems)**

---

5-1
are inserted until the EXU finishes executing the instruction being processed. Then it fetches the next instruction from the instruction queue. When the next instruction is fetched, the BCU advances the state of the bus cycle from state T4 to T1.

When a memory or I/O device has a long access time, the BCU samples the READY signal (sent from memory or an I/O device). If READY is low, the BCU will insert wait states TW between T3 and T4. When READY becomes high, the BCU goes to T4 and then to T1 so that the next instruction can be fetched. When wait state TW is inserted, the current level of each signal is not changed and the read/write timing is longer for that cycle.

Figures 5-1 through 5-8 show read/write timing for μPD70108/70116 memory and I/O. The timing diagrams are for small- and large-scale systems.

**Figure 5-5. Read Timing of μPD70108 Memory and I/O (Large-Scale Systems)**

**Figure 5-6. Write Timing of μPD70108 Memory and I/O (Large-Scale Systems)**

**Figure 5-7. Read Timing of μPD70116 Memory and I/O (Large-Scale Systems)**

**Figure 5-8. Write Timing of μPD70116 Memory and I/O (Large-Scale Systems)**
There are two types of interrupts in the \( \mu P D 7 0 1 0 8 / 7 0 1 1 6 \). One is caused by an external interrupt request and the other is caused internally by software. Both types of interrupts are vectored. When an interrupt occurs, a location in the interrupt vector table is selected either automatically (fixed vector) or by software (variable vector). This selected location determines the start address of the corresponding interrupt routine.

Table 6-1 shows the types of interrupts, interrupt source, number of clocks required to process each interrupt, vector, and priority.

Figure 6-1 shows the interrupt vector table. This table is allocated in a 1 Kbyte memory area (addresses 000H to 3FFH) and can hold up to 256 vectors (four bytes required per vector).

The interrupt sources for vectors 0 to 5 are predetermined and vectors 6 to 31 are reserved for future use. Vectors 32 to 255 are for general use. These vectors are used for the four interrupt sources: 2-byte break, BRKEM, CALLN instructions (during emulation), and INT input.

Four bytes are used for each interrupt vector. The two bytes of the lower address and the two bytes of the higher address are loaded respectively into the program counter (PC) as an offset, and a segment register (PS) as a base.

### Table 6-1. Interrupt Sources

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>No. of Clocks*</th>
<th>Vector</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>External</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NMI (rising-edge triggered)</td>
<td>58/38</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>INT (high-level active)</td>
<td>68/49</td>
<td>32-255</td>
<td>3</td>
</tr>
<tr>
<td>Software</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVU divide by 0 error</td>
<td>65/45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV divide by 0 error</td>
<td>65-75/45-55</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>CHKIND boundary over</td>
<td>81-84/53-56</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>BRKV instruction</td>
<td>60/40</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>BRK3 (breakpoint)</td>
<td>58/38</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>BRK imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRKEM imm8</td>
<td>32-255</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALLN imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRK flag (single step)</td>
<td></td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Note: * The number to the left of the slash (/) is for the \( \mu P D 7 0 1 0 8 \) and the number to the right is for the \( \mu P D 7 0 1 1 6 \).

**Figure 6-1. Interrupt Vector Table**

```
<table>
<thead>
<tr>
<th>Vector</th>
<th>Location</th>
<th>PS</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>00H</td>
<td>(003H, 002H)</td>
<td>(001H, 000H)</td>
</tr>
<tr>
<td>01H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Example: Vector 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The contents of the vectors are initialized at the beginning of a program. The basic steps when program execution jumps to an interrupt routine are:

\[
\begin{align*}
(SP-1, \ SP-2) & \rightarrow PSW \\
(SP-3, \ SP-4) & \rightarrow PS \\
(SP-5, \ SP-6) & \rightarrow PC \\
SP & \rightarrow SP-6 \\
IE & \leftarrow 0, \ BRK & \leftarrow 0, \ MD & \leftarrow 1 \\
PS & \leftarrow \text{higher vector from interrupt vector table} \\
PC & \leftarrow \text{lower vector from interrupt vector table}
\end{align*}
\]

The interrupt enable (IE) and break (BRK) flags are reset when an interrupt routine is started. Therefore, maskable interrupts (INT) and single-step interrupts are disabled.
MASKABLE INTERRUPTS

If an INT input signal is a high level at the end of an instruction and the interrupt is enabled (IE = 1), the INT interrupt request will be acknowledged, unless the NMI or hold request signals are active at the same time. The program execution then enters an interrupt acknowledge cycle (figures 6-2 and 6-3).

The interrupt acknowledge cycle consists of two bus cycles. The INTAK, ASTB, and BUFEN signals are generated during the first cycle. Although the bus cycle is started, no read/write operation is performed and the address/data bus becomes high impedance. During this time, a hold request is not accepted. If the µPD70108/70116 is in the maximum mode, the BUS-...
LOCK signal is also generated inhibiting other devices from using the bus. Figures 6-2 and 6-3 show the timing for the interrupt acknowledge bus cycles.

The first interrupt acknowledge cycle is necessary to synchronize the external interrupt controller with the \( \mu \)PD70108/70116. When the INTAK, ASTB, and BUFEN signals are output during the second interrupt acknowledge cycle, the external interrupt controller puts the interrupt vector number on the data bus (AD\(_7\)-AD\(_0\)).

After the second interrupt acknowledge cycle has been completed, the location in the interrupt vector table corresponding to the vector obtained during the interrupt acknowledge cycle is accessed. Before calling the interrupt routine, the contents of the PSW, PS, and PC are saved in the stack. The interrupt start address is then loaded into the PS and PC registers from the interrupt vector table and the interrupt routine is started.

The following are sequential lists of interrupt acknowledge operations performed by the \( \mu \)PD70108 and \( \mu \)PD70116.

**\( \mu \)PD70108**

1. Acknowledge cycle (first)
2. Acknowledge cycle (second)
3. Save lower byte of PSW to stack
4. Save higher byte of PSW to stack
5. Save lower byte of PS to stack
6. Save higher byte of PS to stack
7. Save lower byte of PC to stack
8. Save higher byte of PC to stack
9. \( \text{SP} \leftarrow \text{SP} - 6 \)
10. Read lower byte of offset word to PC
11. Read higher byte of offset word to PC
12. Read lower byte of segment word to PS
13. Read higher byte of segment word to PS
14. Jump to interrupt start address

**\( \mu \)PD70116**

1. Acknowledge cycle (first)
2. Acknowledge cycle (second)
3. Save PSW word to stack
4. Save PS word to stack
5. Save PC word to stack
6. \( \text{SP} \leftarrow \text{SP} - 6 \)
7. Read offset word to PC
8. Read segment word to PS
9. Jump to interrupt start address

During the first \( \mu \)PD70108 interrupt acknowledge bus cycle, no idle TI states are inserted in the bus cycle. However, the \( \mu \)PD70116 inserts three TI states during the first interrupt acknowledge cycle. During the second interrupt acknowledge cycle, five TI states are inserted in the bus cycles of both microprocessors. Both the \( \mu \)PD70108 and \( \mu \)PD70116 read an 8-bit vector during the second interrupt acknowledge cycle.

The number of cycles required to save the contents of the PSW, PS, and PC are different for the two microprocessors. This is because the width of the \( \mu \)PD70108 data bus is smaller than that of the \( \mu \)PD70116. Two bus cycles are required for the \( \mu \)PD70108 to read the offset word and segment word. Two bus cycles per word are also required to save the PSW, PS, and PC. The \( \mu \)PD70116 performs each of these operations in one bus cycle. The \( \mu \)PD70116 UBE signal remains low during the first and second interrupt acknowledge cycles and during the subsequent accessing of the offset and segment words.

**BRK FLAG (SINGLE-STEP INTERRUPT)**

The \( \mu \)PD70108/70116 is provided with a single-step interrupt function that is useful for program debugging. The Break Flag (bit 8 of the PSW) controls this interrupt. There is no instruction that directly sets or resets the BRK flag; therefore, the PSW must be saved from the stack to control the BRK flag. By restoring the contents of the PSW from the stack, the BRK flag can be set or reset by using OR and AND instructions on the PSW in the stack. When the BRK flag is set, an interrupt routine specified by vector 1 starts after the current instruction has been executed. The BRK and interrupt enable (IE) flags are also reset at this point.

The debug program checks the number of single steps while the interrupt routine is being executed. If the single-step operation can be terminated, a memory operation instruction resets the BRK flag that is saved in the stack. The program then returns to the main routine and the next sequence of instructions is successively carried out. If the program returns to the main routine without changing the BRK flag, the BRK flag (1 in the PSW) will be restored from the stack. The program then executes one instruction of the main routine and the vector 1 interrupt occurs again.
**INTERRUPT DISABLE TIMING**

NMI and INT interrupts are not acknowledged when:

- An instruction that directly sets data in the segment register is being executed; for example
  ```asm
  MOV sreg, reg16
  MOV sreg, mem16
  ```
- The program is between one of the following and the next instruction
  ```asm
  MOV sreg, reg16
  MOV reg16, sreg
  MOV sreg, mem16
  MOV mem16, sreg
  POP sreg
  ```
- Program execution is between one of the following three types of prefix instructions and the next single instruction
  - Segment override prefix (PS:, SS:, DS0:, DS1:)
  - Repeat prefix (REPC, REPNC, REP, REPE, REPZ, REPNE, REPNZ)
  - Bus lock prefix (BUSLOCK)
- Program execution is between the EI instruction and the next instruction (INT only)

Only an NMI request signal generated during the above interrupt disable timing will be internally retained. The request will be acknowledged on completion of the subsequent single instruction.

**INTERRUPTS DURING BLOCK INSTRUCTIONS**

If an external interrupt (NMI or INT with interrupts enabled) occurs while a primitive block transfer, comparison, or I/O instruction is being executed, the CPU will acknowledge the interrupt and branch to the interrupt address. At the beginning of the interrupt routine, the contents of the CW register (a counter for block data) will be saved to the stack. After the contents of the CW have been restored at the end of the interrupt routine, the execution of the CPU will be returned to the original routine. In this manner, the interrupted block operation is resumed.

If prefix instructions have existed before the block operation instruction, up to three will be retained.

When the program returns from the interrupt routine, execution must return to the address at which the prefix instruction is held. For this reason, the \( \mu \text{PD70108/70116} \) modifies the return address (minus one address per prefix instruction) when it is saved.

To best use the \( \mu \text{PD70108/70116} \), do not place more than three prefix instructions before a block operation instruction.

**Correct Example:**

```asm
BUSLOCK
REPC
NMI \rightarrow CMPBK SS: src-block, dst-block
```

In the correct example, the BUSLOCK, REPC, and SS instructions are executed when program execution has been returned from the NMI interrupt process.

**Incorrect Example:**

```asm
BUSLOCK
REP
REPC
NMI \rightarrow CMPBK SS: src-block, dst-block
```

In the incorrect example, only the REP, REPC, and SS instructions will be executed when the program returns from the NMI interrupt process. Since more than three prefix instructions were placed before the block operation instruction, program execution incorrectly returns to the REP instruction instead of the BUSLOCK instruction.
To reset and initialize the μPD70108/70116, a positive pulse must be present on the RESET pin for at least four clock periods.

A CPU reset signal initializes the μPD70108/70116 as follows.

- Clears the following registers to 0000H.
  - PFP (prefetch pointer)
  - PC (program counter)
  - SS (stack segment)
  - DS₀ (data segment 0)
  - DS₁ (data segment 1)
- Sets PS (program segment) register to FFFFH
- Flushes the instruction queue
- Sets or resets the following PSW (program status word) flags:
  - MD = 1 (native mode)
  - DIR = 0 (address direction used during block transfer, Autoincrements)
  - IE = 0 (INT disabled)
  - BRK = 0 (single-step interrupt disabled)

All other registers are undefined.

After the reset signal returns to the low level, the CPU begins execution of the program starting at address FFFF0H.
The \( \mu \)PD70108/70116 has two CPU operating modes: native and 8080 emulation. In native mode, the \( \mu \)PD70108/70116 executes all the instructions given in Section 12, with the exception of the RETEM and CALLN instructions. In 8080 mode, the microprocessor executes the instruction set for the \( \mu \)PD8080AF and the RETEM and CALLN instructions. These modes are selected by special instructions or by using an interrupt. The most significant bit of the PSW is a mode (MD) flag that controls mode selection.

**NATIVE AND 8080 MODE SHIFTING**

When the operating mode is changed from native to emulation or vice versa, the registers will be mapped into the emulation mode as shown in figure 8-1. The lower eight bits of the AW register and both the lower and higher eight bits of the BW, CW, and DW registers of the \( \mu \)PD70108/70116 serve as the accumulator and six general-purpose registers of the \( \mu \)PD8080AF. Figure 8-2 shows the lower eight bits of the PSW of the \( \mu \)PD70108/70116 serving as \( \mu \)PD8080AF flags. These flags correspond to the lower eight bits of the PSW.

The SP register serves as the stack pointer of the \( \mu \)PD8080AF in native mode while the BP register acts as the stack pointer in the emulation mode. In this way, the \( \mu \)PD70108/70116 employs independent stack pointers and stack areas in each mode. Using independent stack pointers prevents destruction of the contents of a stack pointer in one mode due to misoperation of the stack pointer in the other mode. The AH, SP, IX, and IY registers and the four segment registers (PS, SS, DS\(_0\), DS\(_1\)) are not addressable from emulation mode.

In emulation mode, the segment base of the program is determined by the PS register whose contents have been specified by an interrupt vector before the CPU entered emulation mode. The segment base of the memory operands (including the stack) is determined by the DS\(_0\) register whose contents the programmer specifies before the CPU enters emulation mode.

The bus hold function (available by the hold request/acknowledge signal) and standby function (available when the HLT instruction is executed) can be used in emulation mode in the same way as in native mode.

The \( \mu \)PD70108/70116 operates in terms of its normal BCU hardware even in emulation mode. Therefore, I/O operations between the \( \mu \)PD70108/70116 and peripheral circuits or memory are exactly the same as those performed in native mode. However, the BUSLOCK and POLL functions are unavailable for use in emulation mode.

---

**Figure 8.1. Corresponding Registers**

<table>
<thead>
<tr>
<th>Native Mode</th>
<th>8080 Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>AW</td>
<td>ACC</td>
</tr>
<tr>
<td>BW</td>
<td>H</td>
</tr>
<tr>
<td>CW</td>
<td>L</td>
</tr>
<tr>
<td>DW</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>DE</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SP</td>
</tr>
<tr>
<td></td>
<td>BP</td>
</tr>
<tr>
<td></td>
<td>IX</td>
</tr>
<tr>
<td></td>
<td>IY</td>
</tr>
<tr>
<td></td>
<td>PS</td>
</tr>
<tr>
<td></td>
<td>DS(_0)</td>
</tr>
<tr>
<td></td>
<td>SS</td>
</tr>
<tr>
<td></td>
<td>DS(_1)</td>
</tr>
<tr>
<td></td>
<td>PC</td>
</tr>
<tr>
<td></td>
<td>Flags</td>
</tr>
</tbody>
</table>

*User transparent in 8080 mode*

---

**Figure 8.2. Corresponding PSW and Flags**

```
<table>
<thead>
<tr>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE</td>
<td>BRK</td>
<td>S</td>
<td>Z</td>
<td>0</td>
<td>AC</td>
<td>0</td>
<td>P</td>
<td>1</td>
<td>CY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PSW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Z</td>
<td>0</td>
<td>AC</td>
<td>0</td>
<td>P</td>
<td>1</td>
<td>C</td>
<td>Flag</td>
<td></td>
</tr>
</tbody>
</table>
```
To determine externally if the μPD70108/70116 is in emulation mode, confirm that the processor status PS3 signal output during a μPD70108/70116 bus cycle has become high. This signal is always at a low level in native mode. Figure 8-3 shows the mode shift operation of the CPU.

The CPU can reenter emulation mode when INT is present (even if interrupts are disabled) and restart program execution beginning with the instruction after the HLT instruction. This is true only if the CPU entered the standby mode from emulation mode.

If RESET or NMI is present instead of INT — or if INT is present while interrupts are enabled — the CPU will enter native mode from standby mode. If this happens, the CPU can reenter emulation mode from native mode; in other words, from the NMI or INT interrupt routine in native mode, through execution of the RETI instruction. If the CPU entered standby mode from native mode, the CPU can reenter native mode by inputting RESET, NMI, or INT regardless of whether interrupts are disabled or enabled.

**Figure 8-3. Mode Shift Operation of CPU**

**NATIVE TO 8080 EMULATION MODE**

Two instructions cause the operating mode to be changed from native the 8080 emulation mode. These instructions are BRKEM (break for emulation) and RETI (return from interrupt).

**BRKEM imm8 Instruction**

The BRKEM instruction starts the 8080 emulation mode. It saves the contents of the PSW, PS, and PC, and resets the MD flag to 0. The segment base and offset values are then loaded into the PS and PC registers respectively from the interrupt vector table. The interrupt vector number is specified by the immediate operand of the BRKEM instruction.

When the 8080 emulation mode is started by the BRKEM instruction (MD = 0), the CPU executes the program in the 64 Kbyte segment area specified by the contents of the PS, starting from the address indicated by the con-
tents of the PC. The instruction code fetched at this point is interpreted as the PD8080AF instruction and is executed (figure 8-4).

**RETI Instruction**

The RETI instruction is generally used when returning program execution to the main routine from an interrupt routine started by an external interrupt or BRK, or CALLN instruction. When the RETI instruction restores the contents of the PSW, PS, and PC, it also restores the status of the mode (MD) flag before the mode was changed from 8080 to native. This restored MD flag allows the CPU to be returned to the emulation mode again (figure 8-5).

For this reason, if the RETI instruction is executed in native mode at the end of the interrupt routine that has been started by the interrupt instruction CALLN, or by an external interrupt while the CPU is in 8080 mode, the CPU can reenter 8080 mode.

**8080 EMULATION TO NATIVE MODE**

The following signals and instructions are used to change the operating mode from 8080 to native.

- **RESET**
- **NMI or INT**
- **CALLN (call native)**
- **RETEM (return from emulation)**

---

**Figure 8-4. Shift from Native to 8080 Emulation Mode Using BRKEM Instruction**

**Figure 8-5. Shift from Native to 8080 Mode Using RETI Instruction**
**RESET Operation**

When the RESET signal is present, a reset operation is performed on the CPU the same as in native mode. The 8080 emulation in progress is aborted.

**NMI or INT Operation**

When the NMI or INT signal is present, the interrupt process is performed the same as in native mode. Program execution of the CPU will return to the main routine from the interrupt routine in native mode. From native mode, the CPU can reenter the 8080 emulation mode by executing the RETI instruction (figure 8-6).

**CALLN Instruction**

The CALLN instruction is used exclusively in the emulation mode when calling a native mode subroutine not written in 8080 code. If the CALLN instruction is executed in 8080 mode, it causes the CPU to save the contents of the PS, PC, and PSW, and sets the mode flag to 1. This instruction also loads the segment base of an interrupt vector to the segment register (PS) and the offset to the program counter (PC) (figure 8-6).

When the RETI instruction is executed at the end of the interrupt routine, program execution can be returned to the main routine in 8080 emulation mode from the interrupt routine in native mode started by the CALLN instruction.

**RETEM Instruction**

The RETEM instruction is used exclusively as a return from 8080 mode to native mode when the BRKEM instruction caused the shift to the 8080 mode. The RETEM instruction is executed in 8080 emulation mode; program execution of the CPU will return from the BRKEM interrupt routine to the main routine. Consequently, the contents of the PS, PC, and PSW are restored and the CPU reenters native mode. At this time, the MD flag (MD=1), which was saved to the stack by the BRKEM instruction, is restored, causing the CPU to enter native mode (figure 8-7).

---

*Figure 8-6. Shift From 8080 to Native Mode Using NMI, INT, or CALLN Instruction*
EMULATION NESTING

In a native mode called by CALLN or an NMI or INT interrupt from emulation mode, emulation mode cannot be called again by a BRKEM instruction. If this nesting is attempted, MD won't work normally, and normal operation cannot be expected.
The μPD70108/70116 can operate in a standby mode. In standby mode, program execution can be terminated and resumed as required while retaining all internal state information. The clock is not supplied to any circuitry except those required by the hold and standby functions. As a result, power consumption in the standby mode can be reduced to approximately one-tenth of that required for the native or emulation mode. All CPU registers present before standby mode are retained.

**ENTERING STANDBY MODE**

Standby mode is entered whenever the HALT instruction is executed in native or 8080 mode.

**STATUS SIGNALS IN STANDBY MODE**

Although the bus hold function can be used in the standby mode, the CPU reenters the standby mode when the hold acknowledge cycle is completed.

Table 9-1 shows the status of each output signal in standby mode.

<table>
<thead>
<tr>
<th>Table 9-1. Signal Status in Standby Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Output Signal</strong></td>
</tr>
<tr>
<td>Large-scale system mode</td>
</tr>
<tr>
<td>GS1, GS0</td>
</tr>
<tr>
<td>BS2-BS0</td>
</tr>
<tr>
<td>BUSLOCK</td>
</tr>
<tr>
<td>Small-scale system mode</td>
</tr>
<tr>
<td>INTAK</td>
</tr>
<tr>
<td>BUFEN</td>
</tr>
<tr>
<td>WR</td>
</tr>
<tr>
<td>RD</td>
</tr>
<tr>
<td>ASTB</td>
</tr>
<tr>
<td>BUFR/W</td>
</tr>
<tr>
<td>IO/M (μPD70108)</td>
</tr>
<tr>
<td>TO/M (μPD70116)</td>
</tr>
<tr>
<td>LBSO (μPD70108)</td>
</tr>
<tr>
<td>Large- and small-scale system modes</td>
</tr>
<tr>
<td>UBE (μPD70116)</td>
</tr>
<tr>
<td>\text{A}<em>{15}-\text{A}</em>{0} (μPD70108)</td>
</tr>
<tr>
<td>\text{AD}<em>{15}\text{-AD}</em>{0} (μPD70108)</td>
</tr>
<tr>
<td>\text{AD}<em>{15}\text{-AD}</em>{0} (μPD70116)</td>
</tr>
</tbody>
</table>

The control outputs are maintained at inactive levels during the standby mode. The presence of a RESET signal, an external interrupt (NMI or INT), or a bus request from an external bus master will cause the μPD70108/70116 to exit the standby mode.

**EXITING STANDBY MODE BY EXTERNAL INTERRUPTS**

The μPD70108/70116 will exit standby mode when NMI or INT is asserted. When the standby mode is released by an INT signal, the operation the CPU next performs depends upon the state of the IE flag when the HALT instruction is executed.

**Releasing Standby Mode with NMI**

Whether the CPU enters standby mode from the native or emulation mode, the standby mode is unconditionally released when the NMI interrupt is present. If the RETI instruction is executed at the end of the NMI servicing routine, the CPU will reenter the mode that existed before the CPU entered the standby mode. The program is then resumed starting from the instruction which immediately follows the HALT/HLT instruction that caused the standby mode.

**Releasing Standby Mode with INT**

When Interrupts are Disabled (DI). On exiting standby mode, the CPU enters the mode that was set before standby mode. For example, if standby mode was set while the CPU was in native mode, the CPU returns to native mode when it exits standby mode. If the CPU was in emulation mode when standby mode was set, it returns to the 8080 mode. Program execution will be resumed starting from the instruction immediately following the HALT or HLT instruction.

**Note:** When exiting the standby mode by INT (interrupts disabled), INT must be kept at a high level, until the instruction immediately following the HALT/HLT instruction is executed. Therefore, INT must remain at a high level for at least 15 clocks. This assumes the instruction queue is empty after executing the HALT/HLT instruction. If wait states are inserted, the number of inserted wait states must be added to the 15 clocks.

When Interrupts are Enabled (EI). Standby mode is exited when the interrupt routine in native mode is started, regardless of whether the CPU was in native or emulation mode before standby mode was set. If a RETI instruction is executed at the end of the interrupt routine, the CPU will return to the mode that was present just before standby mode was entered. Program execution will start at the instruction immediately following the HALT/HLT instruction.

**EXITING STANDBY MODE BY RESET**

Standby mode is unconditionally exited when RESET becomes active regardless of whether the standby mode was set while the CPU was in native or emulation mode. On exiting the standby mode, a normal CPU reset operation is performed in the native mode.
The μPD70108/70116 has a 20-bit address bus (the lower 8/16 bits are also used as a data bus) and can access up to 1 Mbyte of memory area. The processor employs a memory segment architecture that allows the 1 Mbyte memory area to be treated as logical addresses. The logical addresses are not necessarily the same number as the physical addresses where data is stored.

**PHYSICAL ADDRESS GENERATION**

To obtain a physical address, the contents of a segment register are multiplied by 16 and an offset value known as the "effective address" is then added to the segment register. The result is used as a physical address. The contents of the segment register and the offset value are treated as unsigned data. Also, since the segment register value is multiplied by 16, the segment register may only access physical memory locations which are on a 16 byte boundary; for example, locations 00H, 10H 20H, and so on.

Figure 10-1 shows the relation between a segment register, offset, and physical address.

Using the memory segment method of addressing, you can write programs and only be concerned with the contents of the segment registers and the offset value of the contents. The contents of the segment registers may be a default or specified as an override. If the contents of a segment register constitute address 0, the offsets of the addresses in the segment specified by that segment register can be treated as logical addresses.

A program written as an aggregate of segments specified by logical addresses is compiled, assembled, and treated as object modules. Each object module has its own segment name, size, partition, and control information. These object modules are tied together by the linker and the segment bases corresponding to physical addresses are specified. The object modules can then be loaded into memory.

Unless a specific program is executing an instruction that modifies a segment base — for example, a branch instruction or a variable reference in another segment — the addresses in the program can be determined by the offset from the contents of a segment register. The program can be loaded to any memory area simply by loading the contents of the segment register with the first physical address of the memory area to which the program is to be loaded.

By using segmentation, a program stored in an external file such as a floppy disk can be loaded to any available buffer memory. It will run when the program is called by the program currently being executed by the CPU. In this manner, a program stored in a file or separated into many files can be loaded to an available memory area. This is called "dynamically relocatable code."

**MEMORY SEGMENTS**

Four types of segments are used: Program, Stack, Data 0, and Data 1. The physical address in memory of a segment location is calculated by shifting the value in the segment register to the left four places. An offset value ("effective address") is then added to the shifted segment register value; this sum is the physical address.

The logical segment is specified by one of the four 16-bit registers: PS, SS, DS0, DS1. Each 16-bit register corresponds to one of four logical segments as follows:

<table>
<thead>
<tr>
<th>Segment Register</th>
<th>Default Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS</td>
<td>PFP</td>
</tr>
<tr>
<td>SS</td>
<td>SP, Effective Address</td>
</tr>
<tr>
<td>DS0</td>
<td>IX, Effective Address</td>
</tr>
<tr>
<td>DS1</td>
<td>IY, Effective Address</td>
</tr>
</tbody>
</table>

The function of each segment register is described below.

**Figure 10-1. Physical Addressing**

![Physical Addressing Diagram]
Program Segment
The first address of the program segment is determined by the program segment (PS) register. The offset from the first address is specified by the prefetch pointer (PFP). This segment is primarily used for instruction codes. Data in this segment can be accessed as general variables or source block data by using the segment override prefix (PS:) instruction.

Stack Segment
The first address of the stack segment is determined by the stack segment (SS) register. The offset from the first address is specified by the stack pointer (SP). This segment is used as an area that saves the contents of the PC (return address), PSW, and general purpose registers. The data in the stack segment can be accessed by using the segment override prefix (SS:) instruction.

When addressing the stack, the SS register automatically becomes the segment register if the BP register is specified as the base register. The offset is specified by the effective address.

Data Segment 0
The first address of data segment 0 is determined by the contents of data segment 0 (DS0) register. The offset from the first address is specified by an effective address. When executing a block transfer or BCD string operation instruction, this segment is used to store the source block data. However, the offset is determined then by the contents of the IX register.

When the BP is specified as base register, the default segment register is SS. In this case, you can override with the segment override prefix (DS0:), and the data in data segment 0 can be addressed with DS0 + BP.

Data Segment 1
The first address of data segment 1 is determined by the data segment 1 register (DS1). The offset from the first address is specified by the IY register. This segment is used to store the destination block data when executing a block transfer or BCD string operation instruction. The data in this segment can be accessed as general variables (offset determined by an effective address). The data can also be accessed as source block data (offset determined by the contents of the IX register).
INSTRUCTION ADDRESS

The current address of the μPD70108/70116 program counter (PC) is automatically incremented to the starting location of the next instruction every time the current instruction is about to be executed. In addition, the microprocessor employs the following instruction addressing modes:

- Direct
- Relative
- Register
- Register Indirect
- Indexed
- Based
- Based Index

Direct Addressing

In direct addressing, two bytes of immediate instruction data are directly loaded to the PC or, two bytes are loaded into the PS and two other bytes are loaded into the PC. The immediate data is then used by the PS and PC as a branch address. Direct addressing is used when executing the following instructions:

- CALL far-proc
- CALL memptr16
- CALL memptr32
- BR far-label
- BR memptr16
- BR memptr32

Relative Addressing

In relative addressing, 1 or 2 bytes of immediate instruction data are treated as a signed displacement value and added to the contents of the PC. The result of this addition is the effective address and is used as a branch address.

The sign bit of an 8-bit displacement value is extended and added to the contents of the PC as a 16-bit value. When addition is performed, the contents of the PC indicate the first address of the next instruction.

Relative addressing is used when executing the following instructions:

- CALL near-proc
- CALL near-label
- BR short-label
  Conditional branch instruction short-label

Register Addressing

In register addressing, the contents of any 16-bit register specified by the 3-bit register field in the instruction are loaded to the PC as a branch address. This addressing method allows the use of all eight 16-bit registers (AW, BW, CW, DW, IX, IY, SP, and BP). Register addressing is used when executing the following instructions:

- CALL regptr16
- BR regptr16

Example:

- CALL AW
- BR BW

Register Indirect Addressing

In register indirect addressing, a 16-bit register (IX, IY, or BW) is specified by the register field in an instruction. The specified register then addresses memory.

The addressed contents are then loaded to the PC (or to both the PC and PS) as a branch address.

- CALL memptr16
- CALL memptr32
- BR memptr16
- BR memptr32

Example:

- CALL WORD PTR [IX]
- CALL DWORD PTR [IY]
- BR WORD PTR [BW]
- BR DWORD PTR [IX]

Note: Instruction code memptr16 and memptr32 are generated by the assembler in response to keywords WORD PTR, and DWORD PTR, respectively.

Indexed Addressing

In indexed addressing, 1 or 2 bytes of immediate data in an instruction are treated as a signed displacement value and are added to the contents of a 16-bit register that serves as an index register (IX or IY).

The result of this addition addresses memory and is loaded to the PC as a branch address.

- CALL memptr16
- CALL memptr32
- BR memptr16
- BR memptr32
Example:

```
CALL  var [IX] [2]
CALL  var [IY]
BR    var [IY]
BR    var [IX + 4]
```

**Based Addressing**

In based addressing, 1 or 2 bytes of immediate data in an instruction are treated as a signed displacement value and are added to the contents of a 16-bit register (BP or BW) that serves as a base register. The contents of the memory addressed by the result of this addition are loaded to the PC as an effective address.

Based addressing is used when executing the following instructions:

```
CALL  memptr16
CALL  memptr32
BR    memptr16
BR    memptr32
```

Example:

```
CALL  var [BP + 2]
CALL  var [BP]
BR    var [BW] [2]
BR    var [BP]
```

Note: Instruction code memptr16 is generated by the assembler if variable var has a word attribute. If it has a double word attribute, instruction code memptr32 is generated.

**Based Indexed Addressing**

In based indexed addressing, 1 or 2 bytes of immediate data in an instruction are treated as a signed displacement value. This value is added to the contents of a 16-bit register that serves as a base register (BP or BW) and to the contents of a 16-bit register that serves as an index register (IX or IY). The result of this addition is the effective address. The addressed memory contents are loaded to the PC as a branch address. Based indexed addressing is used when executing the following instructions:

```
CALL  memptr16
CALL  memptr32
BR    memptr16
BR    memptr32
```

Example:

```
CALL  var [BP] [IX]
CALL  var [BW + 2] [IY]
BR    var [BW] [2] [IX]
BR    var [BP + 4] [IY]
```

Note: Instruction code memptr16 is generated by the assembler if variable var has a word attribute. If it has a double word attribute, instruction code memptr32 is generated.

**MEMORY OPERAND ADDRESS**

Several addressing modes and registers are used for particular instruction formats. The memory operand addressing modes are listed below and discussed in the following sections.

- Register
- Immediate
- Direct
- Register Indirect
- Autoincrement/Decrement
- Indexed
- Based
- Based Indexed
- Bit

**Register Addressing**

In register addressing, the contents of the register field (reg = 3-bit field, sreg = 2-bit field) in an instruction addresses a register. See figure 11-1.

**Figure 11-1. Bit Format**

<table>
<thead>
<tr>
<th>Reg</th>
<th>Sreg</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg</td>
<td>Sreg</td>
<td>Item</td>
</tr>
<tr>
<td>reg16</td>
<td>reg8</td>
<td>sreg</td>
</tr>
<tr>
<td>acc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The 3-bit field “reg” is used with bit W of the same instruction and indicates whether a word or a byte register is to be specified. Eight types of word registers (AW, BW, CW, DW, BP, SP, IX, IY) and eight types of byte registers (AL, AH, BL, BH, CL, CH, DL, DH) are specified.

The 2-bit field “sreg” specifies four types of segment registers (PS, SS, DS, and DS1). Sometimes the operation code of an instruction specifies a register. Register addressing is employed when executing instructions that have the following operand formats:

```
CALL  memptr16
CALL  memptr32
BR    memptr16
BR    memptr32
```

Example:

```
When MOV reg, reg is specified:
MOV  BP,SP
MOV  AL,CL
```
Immediate Addressing
In immediate addressing, one or two bytes of immediate data in an instruction are used.

Immediate addressing is used when executing instructions that have the following operand formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm</td>
<td>8/16-bit immediate data</td>
</tr>
<tr>
<td>imm16</td>
<td>16-bit immediate data</td>
</tr>
<tr>
<td>imm8</td>
<td>8-bit immediate data</td>
</tr>
<tr>
<td>pop-value</td>
<td>16-bit immediate data</td>
</tr>
</tbody>
</table>

If imm is specified alone, the assembler checks the value of imm written as an operand or the attribute of other operands that may be written at the same time. The assembler then judges whether the value of imm is 8 or 16 bits. The status of the word/byte specifying bit W is then determined.

Example:
When MOV reg,imm is specified:
  MOV AL, 5 ;Byte — specified by AL.
When MUL reg16, reg16, imm16 is specified:
  MUL AW,BW,1000H ;word — specified by AW and BW.

Direct Addressing
In direct addressing, the immediate data in an instruction addresses memory.

Direct addressing is used when executing the instructions that have the following operand formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>16-bit variable specifying 8 or 16-bit memory data</td>
</tr>
<tr>
<td>dmem</td>
<td>16-bit variable specifying 8 or 16-bit memory data</td>
</tr>
<tr>
<td>imm4</td>
<td>4-bit variable specifying bit length of the bit field data</td>
</tr>
</tbody>
</table>

Example:
When MOV mem,imm is specified:
  MOV WORDVAR, 2000H
When MOV acc,dmem is specified:
  MOV AL, BYTEVAR

Register Indirect Addressing
In register indirect addressing, a 16-bit register (IX, IY, or BW) is determined by the register field in an instruction. The specified register then addresses memory.

Register indirect addressing is used when executing the instructions that have the following operand formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>[IX], [IY], [BW]</td>
</tr>
</tbody>
</table>

Example:
When TEST mem,imm is specified:
  TEST BYTEVAR[IX], 7FH
  TEST BYTEVAR[IX+8], 7FH
  TEST WORDVAR[IX][8], 7FFFH

Note: If variable var has a byte attribute, a byte operand is specified. If it has a word attribute, a word operand is specified. The assembler generates an instruction code to each operand.

Autoincrement/Decrement Addressing
Autoincrement/decrement addressing falls into the category of register indirect addressing.

The contents of a default register addresses a register or memory. Then — if a byte operation is performed — the contents of the default register are automatically incremented/decremented by one. If a word operation is used, the register contents are incremented/decremented by two. The address is automatically modified by this addressing function. This addressing method is always applicable to default registers and is used when executing the instructions that have the following operand formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Default Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>dst-block</td>
<td>IY</td>
</tr>
<tr>
<td>src-block</td>
<td>IX</td>
</tr>
</tbody>
</table>

This addressing will control block data instructions when it is used in combination with counter CW that counts the number of repetitions of the operation.

Indexed Addressing
In indexed addressing, one or two bytes of immediate data in an instruction are treated as a signed displacement value and are added to the contents of a 16-bit register that serves as an index register (IX or IY). The result of this addition forms the effective address used to address a memory operand. Indexed addressing is useful when accessing an array of data. The displacement value indicates the starting address of the array. The contents of the index register determine the address of the data to be accessed.

This addressing method is employed when executing the instructions that have the following operand formats:

Example:
When SUB mem,reg is specified:
  SUB [IX],AW

Note: If variable var has a byte attribute, a byte operand is specified. If it has a word attribute, a word operand is specified. The assembler generates an instruction code to each operand.
Based Addressing

In based addressing, one or two bytes of immediate data in an instruction are treated as a signed displacement value and are added to the contents of a 16-bit base register that serves as a base register (BP or BW). The result of this addition forms the effective address used to address a memory operand.

Based addressing is useful to access structural data that is stored at separate memory locations. The base register indicates the starting address of each structural data and the displacement value selects one piece of data from each structural data.

This addressing method is employed when executing the instructions that have the following operand formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>var[BP],var[BW]</td>
</tr>
<tr>
<td>mem16</td>
<td>var[BP]</td>
</tr>
<tr>
<td>mem8</td>
<td>var[BP]</td>
</tr>
</tbody>
</table>

Example:
When SHL mem,1 is specified:
SHL BYTEVAR[BP],1
SHL WORDVAR[BP+2],1
SHL BYTEVAR[BP] [4],1

Note: If variable var has a byte attribute, a byte operand is specified. If it has a word attribute, a word operand is specified. The assembler generates an instruction code corresponding to each operand.

Based Indexed Addressing

One or two bytes of immediate data in an instruction are treated as a signed displacement value that is added to the contents of two 16-bit registers. One of the registers is a base register (BP or BW) and the other is an index register (IX or IY). The result of the addition forms the effective address that is used to address a memory operand.

Since based indexed addressing allows accessing data by modifying the contents of both the base and index registers, it is useful when accessing arrays of structural data.

For example, the contents of the base register indicate the first address of each structural data. The displacement value in turn indicates the offset from that first address to the first address of a data array. The index register indicates a specific data in the data array.

Based indexed addressing is used when executing instructions that have the following operand formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>var [base register] [index register]</td>
</tr>
<tr>
<td>mem16</td>
<td>var [base register] [index register]</td>
</tr>
<tr>
<td>mem8</td>
<td>var [base register] [index register]</td>
</tr>
</tbody>
</table>

Example:
When PUSH mem16 is specified:
PUSH WORD-VAR [BP] [IX]
PUSH WORD-VAR [BP+2] [IX+6]
PUSH WORD-VAR [BP] [4] [IX] [8]

Bit Addressing

In bit addressing, three or four bits of immediate data in an instruction, or the lower three or four bits of the CL register, specify one bit of an 8 or 16-bit register or memory location.

With bit addressing, a specific single bit in a register or memory can be tested for 0 or 1, set, cleared, or inverted without affecting the other bits. When using the AND or OR instruction to set or reset a bit, a byte or word mask has to be prepared to change one bit. Bit addressing is used when executing the instructions that have the following operand formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm4</td>
<td>Bit number of word operand</td>
</tr>
<tr>
<td>imm3</td>
<td>Bit number of byte operand</td>
</tr>
<tr>
<td>CL</td>
<td>CL</td>
</tr>
</tbody>
</table>

Example:
TEST1 reg8,CL
TEST1 AL,CL
NOT1 reg8,imm3
NOT1 CL,5
CLR1 mem16,CL
CLR1 WORDVAR[IX],CL
SET1 mem16,imm4
SET1 WORDVAR[BP],9
The following sections include instruction formats, descriptions, and examples for the μPD70108/70116 instruction set. For an alphabetical listing by instruction mnemonic, see Appendix A.

The number of clocks assumes the instruction byte(s) have been prefetched and includes the following times:

- Decoding
- EA generation
- Operand fetch
- Execution

The following is a description of the contents of tables 12-1 through 12-7.

<table>
<thead>
<tr>
<th>Table</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>12-1</td>
<td>Identifier and description for the different types of μPD70108/70116 operands</td>
</tr>
<tr>
<td>12-2</td>
<td>Identifiers and descriptions for μPD70108/70116 instruction words</td>
</tr>
<tr>
<td>12-3</td>
<td>Identifier and description of the operations for the μPD70108/70116 instruction set</td>
</tr>
<tr>
<td>12-4</td>
<td>Identifier and description for the different status flags</td>
</tr>
<tr>
<td>12-5</td>
<td>Information about memory addressing, selection of 8- and 16-bit registers, and selection of segment registers.</td>
</tr>
<tr>
<td>12-6</td>
<td>Table 12-1. Operand Types</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td>8- or 16-bit general-purpose register</td>
</tr>
<tr>
<td>reg8</td>
<td>8-bit general-purpose register</td>
</tr>
<tr>
<td>reg16</td>
<td>16-bit general-purpose register</td>
</tr>
<tr>
<td>mem</td>
<td>8- or 16-bit memory location</td>
</tr>
<tr>
<td>mem8</td>
<td>8-bit memory location</td>
</tr>
<tr>
<td>mem16</td>
<td>16-bit memory location</td>
</tr>
<tr>
<td>mem32</td>
<td>32-bit memory location</td>
</tr>
<tr>
<td>dmem</td>
<td>16-bit direct memory address</td>
</tr>
<tr>
<td>imm</td>
<td>8- or 16-bit immediate data</td>
</tr>
<tr>
<td>imm3</td>
<td>3-bit immediate data</td>
</tr>
<tr>
<td>imm4</td>
<td>4-bit immediate data</td>
</tr>
<tr>
<td>imm8</td>
<td>8-bit immediate data</td>
</tr>
<tr>
<td>imm16</td>
<td>16-bit immediate data</td>
</tr>
<tr>
<td>acc</td>
<td>AW or AL accumulator</td>
</tr>
<tr>
<td>sreg</td>
<td>Segment register</td>
</tr>
<tr>
<td>src-table</td>
<td>Name of 256-byte translation table</td>
</tr>
<tr>
<td>src-block</td>
<td>Name of source block addressed by IX register</td>
</tr>
<tr>
<td>dst-block</td>
<td>Name of destination block addressed by IY register</td>
</tr>
<tr>
<td>near-proc</td>
<td>Procedure within the current program segment</td>
</tr>
<tr>
<td>far-proc</td>
<td>Procedure located in another program segment</td>
</tr>
<tr>
<td>near-label</td>
<td>Label in current program segment</td>
</tr>
<tr>
<td>short-label</td>
<td>Label within range of -128 or +127 bytes from end of instruction</td>
</tr>
<tr>
<td>far-label</td>
<td>Label in another program segment</td>
</tr>
<tr>
<td>regptr16</td>
<td>16-bit general-purpose register containing an offset within the current program segment</td>
</tr>
<tr>
<td>memptr16</td>
<td>16-bit memory address containing an offset within the current program segment</td>
</tr>
<tr>
<td>memptr32</td>
<td>32-bit memory address containing the offset and segment data of another program segment</td>
</tr>
<tr>
<td>pop-value</td>
<td>Number of bytes of the stack to be discarded (0-64K, usually even addresses)</td>
</tr>
<tr>
<td>fp-op</td>
<td>Immediate value to identify instruction code of the external floating point processor chip</td>
</tr>
<tr>
<td>R</td>
<td>Register set (AW, BW, CW, DW, SP, BP, IX, IY)</td>
</tr>
<tr>
<td>DS1-spec</td>
<td>(1) DS1</td>
</tr>
<tr>
<td></td>
<td>(2) Segment of group name assumed to DS1</td>
</tr>
<tr>
<td>Seg-spec</td>
<td>(1) Any name or segment register</td>
</tr>
<tr>
<td></td>
<td>(2) Segment or group name assumed to segment register</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional, may be omitted</td>
</tr>
</tbody>
</table>
Table 12-2. Instruction Words

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>Word/Byte specification bit (1 = word, 0 = byte)</td>
</tr>
<tr>
<td>reg</td>
<td>8/16-bit general register specification bit</td>
</tr>
<tr>
<td>mod,mem</td>
<td>Memory addressing specification bits</td>
</tr>
<tr>
<td>(disp-low)</td>
<td>Optional 16-bit displacement lower byte</td>
</tr>
<tr>
<td>(disp-high)</td>
<td>Optional 16-bit displacement higher byte</td>
</tr>
<tr>
<td>disp-low</td>
<td>16-bit displacement lower byte for PC relative</td>
</tr>
<tr>
<td>disp-high</td>
<td>16-bit displacement higher byte for PC relative</td>
</tr>
<tr>
<td>imm3</td>
<td>3-bit immediate data</td>
</tr>
<tr>
<td>imm4</td>
<td>4-bit immediate data</td>
</tr>
<tr>
<td>imm8</td>
<td>8-bit immediate data</td>
</tr>
<tr>
<td>imm16-low</td>
<td>16-bit immediate data lower byte</td>
</tr>
<tr>
<td>imm16-high</td>
<td>16-bit immediate data higher byte</td>
</tr>
<tr>
<td>addr-low</td>
<td>16-bit direct address lower byte</td>
</tr>
<tr>
<td>addr-high</td>
<td>16-bit direct address higher byte</td>
</tr>
<tr>
<td>sreg</td>
<td>Segment register specification bit</td>
</tr>
<tr>
<td>s</td>
<td>Sign-extension specification bit (1 = sign</td>
</tr>
<tr>
<td>way extension, 0 = no sign extension)</td>
<td></td>
</tr>
<tr>
<td>offset-low</td>
<td>Low byte of 16-bit offset data loaded to PC</td>
</tr>
<tr>
<td>offset-high</td>
<td>High byte of 16-bit offset data loaded to PC</td>
</tr>
<tr>
<td>seg-low</td>
<td>Low byte of 16-bit segment data loaded to PS</td>
</tr>
<tr>
<td>pop-value-low</td>
<td>Low byte of 16-bit data which specifies</td>
</tr>
<tr>
<td>pop-value-high</td>
<td>number of bytes of stack to be discarded</td>
</tr>
<tr>
<td>disp8</td>
<td>8-bit displacement added to PC</td>
</tr>
<tr>
<td>X</td>
<td>Operation codes for external floating</td>
</tr>
<tr>
<td>XXX</td>
<td>point processor chip</td>
</tr>
<tr>
<td>YYY</td>
<td></td>
</tr>
<tr>
<td>ZZZ</td>
<td></td>
</tr>
</tbody>
</table>

Table 12-3. Operation Description

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AW</td>
<td>Accumulator (16 bits)</td>
</tr>
<tr>
<td>AH</td>
<td>Accumulator (high byte)</td>
</tr>
<tr>
<td>AL</td>
<td>Accumulator (low byte)</td>
</tr>
<tr>
<td>BW</td>
<td>BW register (16 bits)</td>
</tr>
<tr>
<td>CW</td>
<td>CW register (16 bits)</td>
</tr>
<tr>
<td>CL</td>
<td>CL register (low byte)</td>
</tr>
<tr>
<td>DW</td>
<td>DW register (16 bits)</td>
</tr>
<tr>
<td>SP</td>
<td>Stack pointer (16 bits)</td>
</tr>
<tr>
<td>PC</td>
<td>Program counter (16 bits)</td>
</tr>
<tr>
<td>PSW</td>
<td>Program status word (16 bits)</td>
</tr>
<tr>
<td>IX</td>
<td>Index register (source) (16 bits)</td>
</tr>
<tr>
<td>PS</td>
<td>Program segment register (16 bits)</td>
</tr>
<tr>
<td>DS1</td>
<td>Data segment 1 register (16 bits)</td>
</tr>
<tr>
<td>DS0</td>
<td>Data segment 0 register (16 bits)</td>
</tr>
<tr>
<td>SS</td>
<td>Stack segment register (16 bits)</td>
</tr>
<tr>
<td>AC</td>
<td>Auxiliary carry flag</td>
</tr>
<tr>
<td>CY</td>
<td>Carry flag</td>
</tr>
<tr>
<td>P</td>
<td>Parity flag</td>
</tr>
<tr>
<td>S</td>
<td>Sign flag</td>
</tr>
<tr>
<td>Z</td>
<td>Zero flag</td>
</tr>
<tr>
<td>DIR</td>
<td>Direction flag</td>
</tr>
<tr>
<td>IE</td>
<td>Interrupt enable flag</td>
</tr>
<tr>
<td>V</td>
<td>Overflow flag</td>
</tr>
<tr>
<td>BRK</td>
<td>Break flag</td>
</tr>
<tr>
<td>MD</td>
<td>Mode flag</td>
</tr>
<tr>
<td>(...)</td>
<td>Values in parentheses are memory contents</td>
</tr>
<tr>
<td>disp</td>
<td>Displacement (8 or 16 bits)</td>
</tr>
<tr>
<td>temp</td>
<td>Temporary register (8, 16, or 32 bits)</td>
</tr>
<tr>
<td>seg</td>
<td>Immediate segment data (16 bits)</td>
</tr>
<tr>
<td>offset</td>
<td>Immediate offset data (16 bits)</td>
</tr>
<tr>
<td>←</td>
<td>Transfer direction</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>−</td>
<td>Subtraction</td>
</tr>
<tr>
<td>×</td>
<td>Multiplication</td>
</tr>
<tr>
<td>÷</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulo</td>
</tr>
<tr>
<td>AND</td>
<td>Logical and</td>
</tr>
<tr>
<td>OR</td>
<td>Logical or</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive or</td>
</tr>
<tr>
<td>XXH</td>
<td>2-digit Hexadecimal data</td>
</tr>
<tr>
<td>XXXXXH</td>
<td>4-digit Hexadecimal data</td>
</tr>
</tbody>
</table>
### Table 12-4. Flag Operations

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(blank)</td>
<td>No change</td>
</tr>
<tr>
<td>0</td>
<td>Cleared to 0</td>
</tr>
<tr>
<td>1</td>
<td>Set to 1</td>
</tr>
<tr>
<td>X</td>
<td>Set or cleared according to the result</td>
</tr>
<tr>
<td>U</td>
<td>Undefined</td>
</tr>
<tr>
<td>R</td>
<td>Value saved earlier is restored</td>
</tr>
</tbody>
</table>

### Table 12-5. Memory Addressing

<table>
<thead>
<tr>
<th>mem</th>
<th>mod</th>
<th>00</th>
<th>01</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>BW + IX</td>
<td>BW + IX + disp8</td>
<td>BW + IX + disp16</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>BW + IY</td>
<td>BW + IY + disp8</td>
<td>BW + IY + disp16</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>BP + IX</td>
<td>BP + IX + disp8</td>
<td>BP + IX + disp16</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>BP + IY</td>
<td>BP + IY + disp8</td>
<td>BP + IY + disp16</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>IX</td>
<td>IX + disp8</td>
<td>IX + disp16</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>IY</td>
<td>IY + disp8</td>
<td>IY + disp16</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Direct Address</td>
<td>BP + disp8</td>
<td>BP + disp16</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>BW</td>
<td>BW + disp8</td>
<td>BW + disp16</td>
<td></td>
</tr>
</tbody>
</table>

### Table 12-6. Selection of 8- and 16-Bit Registers

<table>
<thead>
<tr>
<th>reg</th>
<th>W=0</th>
<th>W=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AL</td>
<td>AW</td>
</tr>
<tr>
<td>001</td>
<td>CL</td>
<td>CW</td>
</tr>
<tr>
<td>010</td>
<td>DL</td>
<td>DW</td>
</tr>
<tr>
<td>011</td>
<td>BL</td>
<td>BW</td>
</tr>
<tr>
<td>100</td>
<td>AH</td>
<td>SP</td>
</tr>
<tr>
<td>101</td>
<td>CH</td>
<td>BP</td>
</tr>
<tr>
<td>110</td>
<td>DH</td>
<td>IX</td>
</tr>
<tr>
<td>111</td>
<td>BH</td>
<td>IY</td>
</tr>
</tbody>
</table>

### Table 12-7. Selection of Segment Registers

<table>
<thead>
<tr>
<th>reg</th>
<th>sreg</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>DS1</td>
</tr>
<tr>
<td>01</td>
<td>PS</td>
</tr>
<tr>
<td>10</td>
<td>SS</td>
</tr>
<tr>
<td>11</td>
<td>DS0</td>
</tr>
</tbody>
</table>
DATA TRANSFER

MOV reg, reg

Move register to register

```
<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

reg ← reg

Transfers the contents of the 8- or 16-bit register specified by the second operand to the 8- or 16-bit register specified by the first operand.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation: None

Example:

- MOV BP, SP
- MOV AL, CH

MOV mem, reg

Move register to memory

```
<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
mod reg mem
```

```
(disp-low)
```

```
(disp-high)
```

(mem) ← reg

Transfers the contents of the 8- or 16-bit register specified by the second operand to the 8- or 16-bit memory location specified by the first operand.

Bytes: 2/3/4
Clocks:

When W = 0: 9
When W = 1: 13, µPD70108
13, µPD70116 odd addresses
9, µPD70116 even addresses

Transfers: 1
Flag operation: None

Example:

- MOV [BP][IX], AW
- MOV BYTE_VAR, BL
**MOV reg,mem**
Memory to register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>W</td>
</tr>
</tbody>
</table>

mod  reg  mem

<table>
<thead>
<tr>
<th></th>
<th>(disp-low)</th>
<th>(disp-high)</th>
</tr>
</thead>
</table>

reg ← (mem)
Transfers the 8- or 16-bit memory contents specified by the second operand to the 8- or 16-bit register specified by the first operand.

Bytes: 2/3/4

Clocks:
When W = 0: 11
When W = 1: 15, µPD70108
15, µPD70116 odd addresses
11, µPD70116 even addresses

Transfers: 1
Flag operation: None

Example:
MOV  AW,[BW][IY]
MOV  CL,BYTE_VAR

**MOV mem,imm**
Immediate data to memory

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>W</td>
</tr>
</tbody>
</table>

mod  0  0  0  mem

<table>
<thead>
<tr>
<th></th>
<th>(disp-low)</th>
<th>(disp-high)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>imm8- or imm16-low</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>imm16-high</td>
</tr>
</tbody>
</table>

(mem) ← imm
Transfers the 8- or 16-bit immediate data specified by the second operand to the 8- or 16-bit memory location addressed by the first operand.

Bytes: 3/4/5/6

Clocks:
When W = 0: 11
When W = 1: 15, µPD70108
15, µPD70116 odd addresses
11, µPD70116 even addresses

Transfers: 1
Flag operation: None

Example:
MOV  BYTE_PTR [BP][IX],0
MOV  WORD_PTR [BW],12
MOV  [BP][IX],5  ;Note: assembler assumes WORD_PTR as default.
MOV  BYTE_VAR,123
MOV  WORD_VAR,1000H
MOV reg,imm
Immediate data to register

\[
\begin{array}{cccc|c}
7 & 3 & 2 & 0 & \text{reg} \\
1 & 0 & 1 & 1 & W \text{ imm8- or imm16-low} \\
   &   &   &   & \text{imm16-high} \\
\end{array}
\]

\text{reg} \leftarrow \text{imm}
Transfers the 8- or 16-bit immediate data specified by the second operand to the 8- or 16-bit register specified by the first operand.

Bytes: 2/3
Clocks: 4
Transfers: None
Flag operation: None
Example: MOV BP,8000H

MOV acc,dmem
Memory to accumulator

\[
\begin{array}{cccccc|c}
7 & 3 & 2 & 0 & \text{addr-low} & \text{addr-high} & \text{W} \\
1 & 0 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

When \( W = 0 \) \( \text{AL} \leftarrow (\text{dmem}) \)
When \( W = 1 \) \( \text{AH} \leftarrow (\text{dmem} + 1), \text{AL} \leftarrow (\text{dmem}) \)
Transfers the memory contents addressed by the second operand to the accumulator (AL or AW) specified by the first operand.

Bytes: 3
Clocks:
When \( W = 0 \): 10
When \( W = 1 \): 14: \( \mu \text{PD70108} \)
\( \mu \text{PD70116 odd addresses} \)
10: \( \mu \text{PD70116 even addresses} \)
Transfers: 1
Flag operation: None
Example:

MOV AW,WORD_VAR
MOV AL,BYTE_VAR
### MOV dmem, acc

**Accumulator to memory**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>W</td>
</tr>
</tbody>
</table>

- **addr-low**
- **addr-high**

When \( W = 0 \), \((dmem) \leftarrow AL\)
When \( W = 1 \), \((dmem + 1) \leftarrow AH, (dmem) \leftarrow AL\)

Transfers the contents of the accumulator (AL or AW) specified by the second operand to the 8- or 16-bit memory location addressed by the first operand.

<table>
<thead>
<tr>
<th>Bytes:</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clocks:</td>
<td></td>
</tr>
<tr>
<td>When ( W = 0 ):</td>
<td>9</td>
</tr>
</tbody>
</table>
| When \( W = 1 \): | 13, \(\mu P D 7 0 1 0 8\)  
13, \(\mu P D 7 0 1 1 6\) odd addresses  
9, \(\mu P D 7 0 1 1 6\) even addresses |
| Transfers: | 1 |
| Flag operation: | None |

**Example:**
- MOV WORD_VAR, AW
- MOV BYTE_VAR, AL

### MOV sreg, reg16

**Register to segment register**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **sreg \leftarrow reg16 sreg**: SS, DS0, DS1

Transfers the contents of the 16-bit register specified by the second operand to the segment register (except PS) specified by the first operand. External interrupts (NMI, INT) or a single-step break is not accepted between this instruction and the next.

<table>
<thead>
<tr>
<th>Bytes:</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clocks:</td>
<td>2</td>
</tr>
<tr>
<td>Transfers:</td>
<td>None</td>
</tr>
<tr>
<td>Flag operation:</td>
<td>None</td>
</tr>
</tbody>
</table>

**Example:**
- MOV SS, AW
MOV sreg,mem16
Memory to segment register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

mod | sreg | mem

(disp-low)

(disp-high)

sreg ← (mem16) sreg: SS,DS0,DS1
Transfers the 16-bit memory contents addressed by the second operand to the segment register (except PS) specified by the first operand. However, external interrupts (NMI, INT) or a single-step break is not accepted during the period between this instruction and the next.

Bytes: 2/3/4
Clocks:
  When W = 0: 11
  When W = 1: 15, μPD70108
  15, μPD70116 odd addresses
  11, μPD70116 even addresses

Transfers: 1
Flag operation: None

Example:
  MOV DS0,[BW][IX]
  MOV SS,WORD_VAR

MOV reg16,sreg
Segment register to register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

reg 16 ← sreg
Transfers the contents of the segment register specified by the second operand to the 16-bit register specified by the first operand.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation: None

Example: MOV AW,DS1
MOV mem16,sreg
Segment register to memory

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

mod 0 sreg mem

(disp-low)

(disp-high)

(mem16) ← sreg

Transfers the contents of the segment register specified by the second operand to the 16-bit memory location addressed by the first operand.

Bytes: 2/3/4

Clocks:
- When W = 0: 10
- When W = 1: 14, µPD70108
  14, µPD70116 odd addresses
  10, µPD70116 even addresses

Transfers: 1

Flag operation: None

Example:
MOV [IX],PS

MOV DS0,reg16,mem32
32-bit memory to 16-bit register and DS0

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

mod reg mem

(disp-low)

(disp-high)

reg 16 ← (mem32)
DS0 ← (mem32 + 2)

Transfers the lower 16 bits (offset word of a 32-bit pointer variable) addressed by the third operand to the 16-bit register specified by the second operand, and the higher 16 bits (segment word) to the DS0 segment register.

Bytes: 2/3/4

Clocks:
- 26, µPD70108
- 26, µPD70116 odd addresses
- 18, µPD70116 even addresses

Transfers: 2

Flag operation: None

Example: MOV DS0,BW,DWORD_VAR
MOV DS1,reg16,mem32
32-bit memory to 16-bit register and DS1

| 7 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |

mod    reg    mem

(disp-low)

(disp-high)

reg16 ← (mem32)
DS1 ← (mem32 + 2)

Transfers the lower 16 bits (offset word of a 32-bit pointer variable) addressed by the third operand to the 16-bit register specified by the second operand, and the higher 16 bits (segment word) to the DS1 segment register.

Bytes: 2/3/4
Clocks:
26, μPD70108
26, μPD70116 odd addresses
18, μPD70116 even addresses

Transfers: 2
Flag operation: None
Example: MOV DS1,IY,DWORD_VAR

MOV AH,PSW
PSW to AH

| 7 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

AH ← S,Z,X,AC,X,P,X,CY

Transfers flags S, Z, AC, P, and CY of PSW to the AH register. Bits 5, 3, and 1 are undefined.

Bytes: 1
Clocks: 2
Transfers: None
Flag operation: None
Example: MOV AH,PSW
MOV  PSW,AH

AH to PSW

\[
\begin{array}{ccccccc}
7 & & & & & & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 0
\end{array}
\]

S,Z,X,AC,X,P,X,CY ← AH

Transfers bits 7, 6, 4, 2, 0 of the AH register to flags S, Z, AC, P, and CY of PSW.

Bytes: 1

Clocks: 3

Transfers: None

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: MOV  PSW,AH

LDEA reg16, mem16

Load effective address to register

\[
\begin{array}{ccccccc}
7 & & & & & & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 1
\end{array}
\]

mod  reg  mem

\[
\begin{array}{c}
\text{(disp-low)}
\end{array}
\]

\[
\begin{array}{c}
\text{(disp-high)}
\end{array}
\]

reg16 ← mem16

Loads the effective address (offset) generated by the second operand to the 16-bit general-purpose register specified by the first operand. Used to set starting address values to the registers that automatically specify the operand for TRANS or block instructions.

Bytes: 2/3/4

Clocks: 4

Transfers: None

Flag operation: None

Example: LDEA  BW, TABLE[IX]
TRANS no operand
TRANS src-table
TRANSB no operand

Translate byte

\[
\begin{array}{cccccc}
7 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
\end{array}
\]

AL ← (BW + AL)

Transfers to the AL register one byte specified by the BW and AL registers from the 256-byte conversion table. This time, the BW register specifies the starting (base) address of the table, while the AL register specifies the offset value within 256 bytes of the starting address.

Bytes: 1
Clocks: 9
Transfers: 1
Flag operation: None

Example:

TRANS TABLE
TRANS
TRANSB

XCH reg,reg

Exchange register with register

\[
\begin{array}{ccccccc}
7 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & W \\
\end{array}
\]

reg ↔ reg

Exchanges the contents of the 8- or 16-bit register specified by the first operand with the contents of the 8- or 16-bit register specified by the second operand.

Bytes: 2
Clocks: 3
Transfers: None
Flag operation: None

Example:

XCH CW,BW
XCH AH,AL
**Section 12**
*Instruction Set*

**XCH mem, reg**
**XCH reg, mem**

Exchange memory with register

```
<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
mod  reg  mem
```

```
(disp-low)
```

```
(disp-high)
```

(mem) ↔ reg

Exchanges the 8- or 16-bit memory contents addressed by the first operand with the contents of the 8- or 16-bit register specified by the second operand.

**Bytes:** 2/3/4

**Clocks:**
- When \( W=0 \): 16
- When \( W=1 \): 24, \( \mu P D 7 0 1 0 8 \)
  - 24, \( \mu P D 7 0 1 1 6 \) odd addresses
  - 16, \( \mu P D 7 0 1 1 6 \) even addresses

**Transfers:** 2

**Flag operation:** None

**Example:**

XCH WORD_VAR, CW
XCH AL, TABLE[BW]

**XCH AW, reg16**
**XCH reg16, AW**

Exchange accumulator with register

```
<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
AW ↔ reg16
```

Exchanges the contents of the accumulator (AW only) specified by the first operand with the contents of the 16-bit register specified by the second operand.

**Bytes:** 1

**Clocks:** 3

**Transfers:** None

**Flag operation:** None

**Example:**

XCH AW,DW
XCH CW, AW
REPEAT PREFIXES

REPC (no operand)
Repeat while carry

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

While CW ≠ 0, the block comparison instruction (CMPBK or CMPM) placed in the following byte is executed and CW is decremented (−1). If the result of the block comparison instruction is CY ≠ 1, the instruction terminates. CW is checked against the condition immediately before the execution of the block comparison instruction. Therefore, if CW = 0 the first time the REPC instruction is executed, the program will proceed immediately to the instruction following the block comparison instruction and the block comparison instruction will not be executed at all. The contents of CY immediately before the first execution of the REPC instruction are “don’t care.”

Bytes: 1
Clocks: 2
Transfers: None
Flag operation: None
Example: REPC CMPBK

REPNC (no operand)
Repeat while no carry

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

While CW ≠ 0, the block comparison instruction (CMPBK or CMPM) placed in the following byte is executed and CW is decremented (−1). If the result of the comparison instruction is CY = 1, the instruction terminates. CW is checked against the condition immediately before the execution of the block comparison instruction. Therefore, if CW = 0 the first time the REPNC instruction is executed, the program will proceed immediately to the instruction following the block comparison instruction and the block comparison instruction will not be executed at all. The contents of CY immediately before the first execution of the REPNC instruction are “don’t care.”

Bytes: 1
Clocks: 2
Transfers: None
Flag operation: None
Example: REPNC CMPMB
REP/REPE/REPZ
Repeat/repeat while equal/repeat while zero
REP (no operand)
REPE/REPZ (no operand)

While CW ≠ 0, the following instruction is executed and CW is decremented (−1).

REP is used with MOVBK, LDM, STM, OUTM, or INM instructions and performs repeat operations while CW ≠ 0. The Z flag is disregarded.

REPE or REPZ is used with the CMPBK or CMPM instruction. A program will exit the loop if the comparison result by each block instruction is Z ≠ 1 or when CW becomes 0.

CW is checked against the condition immediately before the execution of REP/REPE/REPZ instruction. Consequently, if CW = 0 the first time the REP/REPE/REPZ instruction is executed, the program will move to the instruction following the block instruction and the block instruction will not be executed at all.

A zero flag check is performed against the result of the block instruction. The contents immediately before the first execution of the REPE/REPZ instruction are “don’t care.”

Bytes: 1
Clocks: 2
Transfers: None
Flag operation: None
Example:
REP MOVBK
REPE CMPBK
REPE CMPMB

REPNE/REPNZ (no operand)
Repeat while not equal/repeat while not zero

While CW ≠ 0, the block comparison instruction (CMPBK, CMPM) is executed and CW is decremented (−1). If the result of the block comparison instruction is Z ≠ 0 or CW becomes 0, the instruction terminates. CW is checked against the condition immediately before the execution of the block comparison instruction. Consequently, if CW = 0 the first time the REPNE/REPNZ instruction is executed, the program will proceed immediately to the instruction following the block comparison instruction, and the block comparison instruction will not be executed at all.

A zero flag check is performed to test the result of the block comparison instruction. The contents of Z immediately before the first execution of the REPNE/REPNZ instruction are “don’t care.”

Bytes: 1
Clocks: 2
Transfers: None
Flag operation: None
Example:
REPNE CMPMB
REPNZ CMPBK
**PRIMITIVE BLOCK TRANSFER**

MOV BK/ MOV KB/ MOV BKW

(repeat) MOV BK [DS1-spec:] dst-block,[Seg-spec:] src-block

(repeat) MOV KBK (no operand)

(repeat) MOV BKW (no operand)

**Move block/move block byte/move block word**

7

| 1 | 0 | 1 | 0 | 0 | 1 | 0 | W |

When \( W = 0 \), \((IY) \leftarrow (IX)\)

- \( \text{DIR} = 0 \): \( IX \leftarrow IX + 1, IY \leftarrow IY + 1 \)
- \( \text{DIR} = 1 \): \( IX \leftarrow IX - 1, IY \leftarrow IY - 1 \)

When \( W = 1 \), \((IY + 1, IY) \leftarrow (IX + 1, IX)\)

- \( \text{DIR} = 0 \): \( IX \leftarrow IX + 2, IY \leftarrow IY + 2 \)
- \( \text{DIR} = 1 \): \( IX \leftarrow IX - 2, IY \leftarrow IY - 2 \)

Transfers the block addressed by the IX register to the block addressed by the IY register by repeating the data word byte. In order to transfer the next byte/word, the IX or IY register is automatically incremented \((+1 \text{ or } +2)\) or decremented \((-1 \text{ or } -2)\) each time a byte/word is transferred. The direction of the block is determined by the direction flag (DIR).

Byte or word specification is made by the attribute of the operand when the MOV BK is used. If the MOV KBK or MOV BKW is used, the type is specified by the instruction.

The destination block must always be located within the segment specified by the DS1 segment register. The default segment for the source block register is DS0, and a segment override is permitted. The source block may be located in a segment specified by any of the segment registers.

**Bytes:** 1

**Clocks:**

**Repeat:**

- When \( W = 0 \): \( 11 + 8 / \text{rep} \)
- When \( W = 1 \): \( 11 + 16 / \text{rep}, \mu PD70108 \)
  - \( \mu PD70116 \) odd, odd addresses
  - \( 11 + 16 / \text{rep}, \mu PD70116 \) odd, even addresses
  - \( 11 + 8 / \text{rep}, \mu PD70116 \) even, even addresses

**Single operation:**

- When \( W = 0 \): \( 11 \)
- When \( W = 1 \): \( 19, \mu PD70108 \)
  - \( 19, \mu PD70116 \) odd, odd addresses
  - \( 15, \mu PD70116 \) odd, even addresses
  - \( 11, \mu PD70116 \) even, even addresses

**Transfers:**

- Repeat: \( 2 / \text{rep} \)
- Single operation: \( 2 \)

**Flag operation:** None

**Examples:**

1. MOV AW, SEG SRC BLOCK
   ; point to source
   MOV DS0, AW
   ; segment and offset
   MOV IX, OFFSET SRC BLOCK
   MOV AW, SEG DST BLOCK
   ; point to destination
   MOV DS1, AW
   MOV IY, OFFSET DST BLOCK
   MOV CW, 22
   ; set count
   REP MOV BKW
   ; move 22 words

2. MOV IX, SP
   ; source will be stack
   MOV DS1, IY, DST DWPTR
   ; fetch pointer to destination
   MOV CW, 5
   ; set count
   REP MOV BK DS1, DST BLOCK, SS: [IX]
   ; move from stack (override prefix)
   ; to destination

**DATA0 SEGMENT AT 0**

- SRC BLOCK: DW 22 DUP (?)
- SRC DWPTR: DD SRC BLOCK
- DST DWPTR: DD DST BLOCK

**DATA0 ENDS**

**DATA1 SEGMENT AT 1000H**

- DST BLOCK: DW 22 DUP (?)

**DATA1 ENDS**

12-16
### CMPBK/COMPBKB/CMPBKW

**Repeat** CMPBK [Seg-spec:]src-block,[DS1-spec:]dst-block

**Repeat** CMPBKB (no operand)

**Repeat** CMPBKW (no operand)

Compare block/compare block byte/compare block word

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### When W=0:

**DIR=0:** IX ← IX+1, IY ← IY+1

**DIR=1:** IX ← IX–1, IY ← IY–1

#### When W=1:

**DIR=0:** (IX+1, IX) ← (IY+1, IY)

**DIR=1:** IX ← IX–2, IY ← IY–2

Repeatedly compares the block addressed by the IX register with the block addressed by the IY register, byte by byte or word by word. The result of the comparison is shown by the flag. In order to process the next byte or word, IX and IY are automatically incremented (+1 or +2) or decremented (−1 or −2) each time one byte or word is processed. The direction of the block is determined by the direction flag (DIR).

The byte or word specification is made by the attribute of the operand when CMPBK is used. If CMPBKB or CMPBKW is used, it is specified directly to be the byte or word type.

The destination block must always be located within the segment specified by the DS1 register. The default segment register for the source block is DS0 and a segment override prefix is permitted.

**Bytes:** 1

**Clocks:**

**Repeat:**

When W=0: 7+14/rep

When W=1: 7+22/rep

#### Single operation:

| When W=0: | 13 |
| When W=1: | 21 |
| μPD70108 | μPD70116 odd, odd addresses |
| 17: | μPD70116 odd, even addresses |
| 13: | μPD70116 even, even addresses |

**Transfers:**

Repeat: 1/rep

**Single operation:** 2

**Flag operation**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

MOV DS0, IX, SRC_DWPTR  ; point to areas to compare
MOV DS1, IY, DST_DWPTR
MOV CW, 16  ; set count
REPNC CMPBKB  ; compare 16 pairs of bytes
BCWZ GREATER  ; if CW = 0, then SRC ≥ DST

**LESS:** ---
CMPM/CMPMB/CMPMW

(repeat) CMPM [DS1-spec:]dst-block
(repeat) CMPMB (no operand)
(repeat) CMPMW (no operand)

Compare multiple/compute multiple byte / compare multiple word

\[
\begin{array}{ccccccc}
7 & 1 & 0 & 1 & 0 & 1 & 1 \ 0 & W
\end{array}
\]

When W=0:
- (AL) - (IY)
- DIR=0: IY ← IY+1
- DIR=1: IY ← IY - 1
When W=1:
- AW - (IY+1, IY)
- DIR=0: IY ← IY+2
- DIR=1: IY ← IY-2

Repeatedly compares the block addressed by the IY with the accumulator (AL or AW). To process the next byte or word, the IY is automatically incremented (+1 or +2) or decremented (-1 or -2) each time one byte or word is processed. The direction of the block is determined by the direction flag (DIR). Byte or word specification is made by the attribute of the operand when CMPM is used. If CMPMB or CMPMW is used, it is specified directly by the instruction.

The destination block must always be located within the segment specified by the DS1 segment register.

Bytes: 1

Clocks:
Repeat:
- When W=0: 7+10/rep
- When W=1: 7+14/rep, \( \mu PD70108 \)
- 7+14/rep, \( \mu PD70116 \) odd addresses
- 7+10/rep, \( \mu PD70116 \) even addresses

Single operation:
- When W=0: 7
- When W=1: 11, \( \mu PD70108 \)
- 11, \( \mu PD70116 \) odd addresses
- 7, \( \mu PD70116 \) even addresses

Transfers:
- Repeat: 1/rep
- Single operation: 1

Flag operation

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

- MOV DS1,IY,DST,DWPTR
  ;point to destination block
- MOV AL,'A'
- MOV CW,20
  ;search for first 'A'
- REPNZ CMPMB
LDM/LDMB/LDMW
(repeat) LDM [Seg-spec:]src-block
(repeat) LDMB (no operand)
(repeat) LDMW (no operand)

Load multiple/load multiple byte/load multiple word

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

When W=0:
AL ← (IX)
DIR=0: IX ← IX+1
DIR=1: IX ← IX-1

When W=1:
AW ← (IX+1, IX)
DIR=0: IX ← IX+2
DIR=1: IX ← IX-2

Transfers the block addressed by the IX register to the accumulator (AL or AW). To process the next byte or word the IX register is automatically incremented (+1 or +2) or decremented (−1 or −2) each time one byte or word is processed. The direction of the block is determined by the direction flag (DIR). Byte or word specification is made by the attribute of the operand when LDM is used. If LDMB or LDMW is used, it is specified directly to be the byte or word type. The instruction may have a repeat prefix, but is usually used without one.

The default segment register for the source block is DS0, and therefore segment override is possible. The source block may be located within the segment specified by any (optional) segment register.

Bytes: 1

Clocks:
Repeat:
When W=0: 7+9/rep
When W=1: 7+13/rep: μPD70108
7+13/rep: μPD70116 odd addresses
7+9/rep: μPD70116 even addresses

Single operation:
When W=0: 7,
When W=1: 11, μPD70108
11, μPD70116 odd addresses
7, μPD70116 even addresses

Transfers:
Repeat: 1/rep
Single operation: 1

Flag operation: None

Example:

; Add a constant to a string
MOV DS1:IY, DST_DWPTR
; point DS1:IY to string
MOV IX, IY
; point DS1:IX to same area
MOV CW, 10
; length of string
HERE: LDM BYTE PTR DS1:[IX]
; fetch byte (from DS1, with segment override prefix), increment IX
ADD AL, 20H
; add constant
STMB ; replace modified value at DS1:IY,
; increment IY
DBNZ HERE ; loop until CW = 0
STM/STMB/STMW
(repeat) STM [DS1-spec:]dst-block
(repeat) STMB (no operand)
(repeat) STMW (no operand)

Store multiple/store multiple byte/store multiple word

\[
\begin{array}{cccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & W
\end{array}
\]

When \(W=0\):
- \((IY) \leftarrow AL\)
- \(DIR=0: \ IY \leftarrow IY+1\)
- \(DIR=1: \ IY \leftarrow IY-1\)

When \(W=1\):
- \((IY+1, IY) \leftarrow AW\)
- \(DIR=0: \ IY \leftarrow IY+2\)
- \(DIR=1: \ IY \leftarrow IY-2\)

Transfers the contents of \(AL\) or \(AW\) to the block addressed by \(IY\).

To process the next byte or word, \(IY\) is automatically incremented (+1 or +2) or decremented (−1 or −2) each time one byte or word is processed. The direction of the block is determined by the direction flag (\(DIR\)).

Byte or word specification is made by the attribute of the operand when STM is used. If STMB or STMW is used, it is specified directly to be the byte or word type.

The destination block must always be located within the segment specified by the DS1 segment register.

Bytes: 1
Clocks:
Repeat:
- When \(W=0\): 7+4/rep
- When \(W=1\): 7+8/rep: \(\mu\)PD70108
  - 7+8/rep: \(\mu\)PD70116 odd addresses
  - 7+4/rep: \(\mu\)PD70116 even addresses

Single operation:
- When \(W=0\): 7
- When \(W=1\): 11, \(\mu\)PD70108
  - 11, \(\mu\)PD70116 odd addresses
  - 7, \(\mu\)PD70116 even addresses

Transfers:
Repeat: 1/rep
Single operation: 1

Flag operation: None
BIT FIELD MANIPULATION INSTRUCTIONS

- **INS reg1, reg2**

  Insert bit field (register)

  16-bit field ← AW

  Transfers the lower data bits of the 16-bit AW register (bit length is specified by the 8-bit register of the second operand) to the memory location determined by the byte offset (addressed by the DS1 segment register and the IY index register) and bit offset (specified by the 8-bit register of the first operand).

  After the transfer, the IY register and the 8-bit register specified by the first operand are automatically updated to point to the next bit field.

  Only the lower 4 bits (0-15) will be valid for the 8-bit register of the first operand that specifies the bit offset (maximum length: 15 bits). Also, only the lower 4 bits (0-15) will be valid for the 8-bit register of the second operand that specifies the bit length (maximum length: 16 bits). 0 specifies a 1-bit length, and 15 specifies a 16-bit length.

  Bit field data may overlap the byte boundary of memory. Note: For correct operation the upper four bits of the 8-bit registers used as first and second operands must be set to 0.

  Bytes: 3
  
  Clocks:
  35-113: μPD70108
  35-113: μPD70116 odd addresses
  31-117: μPD70116 even addresses
  
  Transfers: 2 or 4
  
  Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

  Example: INS DL,CL (See below for detailed example)
INS reg8,imm4
Insert bit field (immediate data)

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>imm4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

16-bit field ← AW

Transfers the lower data bits of the 16-bit AW register (bit length specified by the 4-bit immediate data of the second operand) to the memory location determined by the byte offset (addressed by the DS1 segment register and the IY register) and bit offset (specified by the 8-bit register of the first operand). After the transfer, the IY register and the 8-bit register specified by the first operand are updated to point to the next bit field.

Only the lower 4 bits (0-15) for the 8-bit register of the first operand (15 bits maximum length) are valid. The immediate data value of the second operand (16 bits maximum length) is valid only from 0-15.

0 specifies a 1-bit length, and 15 specifies a 16-bit length. The bit field data may overlap the byte boundary of memory.

Note: For correct operation, set the upper four bits of the 8-bit register used as the first operand to 0.

Bytes: 4
Clocks:
75-103: µPD70108
75-103: µPD70116 odd addresses
67-87: µPD70116 even addresses

Transfers: 2 or 4
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

Example:

```
MOV DS1,IY,DST,DWPTR ;Point to destination
MOV CL,3            ;Start at bit 3
MOV DL,4            ;Insert 5 bits
(A) MOV AW,5555H    ;Pattern to insert (A)
(B) INS CL,DL      ;Insert 5 bits at bit 3 (B)
(C) INS CL,12      ;Insert 13 bits at bit 8 (C)
```

- at (A) memory =
  MSB | LSB
  XXXX XXXX XXXX XXXX | XXXX XXXX XXXX XXXX
  CL = 3, IY = base

- at (B) memory =
  » XXXX XXXX XXXX XXXX | XXXX XXXX 1010 1XXX
  CL = 8, IY = base

- at (C) memory =
  » XXXX XXXX XXXX1 0101 | 0101 0101 1010 1XXX
  CL = 5, IY = base + 2
EXT reg1, reg2

Extract bit field (register)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

AW ← 16-bit field

Loads the bit field data (bit length specified by the 8-bit register of the second operand) into the AW register. The segment base of the memory location of the bit field is specified by the DS0 register, the byte offset by the IX index register, and the bit offset by the 8-bit register of the first operand. At the same time zeros are loaded to the remaining upper bits of the AW register.

After the transfer, the IX register and the 8-bit register specified by the first operand are updated to point to the next bit field. Only the lower 4 bits (0-15) of the 8-bit register of the first operand (maximum length: 15 bits) are valid. Only the lower 4 bits of the 8-bit register of the second operand (maximum length: 16 bits) are valid.

0 specifies a 1-bit length, and 15 specifies a 16-bit length. Bit field data may overlap the byte boundary of memory.

Note: For correct operation, the upper 4 bits of the 8-bit registers used as first and second operands must be set to 0.

Bytes: 3

Clocks:
34-59: μPD70108
34-59: μPD70116 odd addresses
26-55: μPD70115 even addresses

Transfers: 1 or 2

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

Example: EXT CL,DL (See below for detailed example)
EXT reg8,imm4

Extract bit field (immediate data)

<table>
<thead>
<tr>
<th>reg8</th>
<th>imm4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

AW ← 16-bit field

Loads bit field data from the memory location specified by the byte offset to the AW register (addressed by the DS0 segment register and the IX index register) and the bit offset (specified by the 8-bit register of the first operand).

The bit length is specified by the 4-bit immediate data of the second operand.

After the transfer, the IX register and the 8-bit register specified by the first operand are updated to point to the next bit field. Only the lower 4 bits (0-15) of the 8-bit register of the first operand (maximum length: 15 bits) will be valid. The immediate data value of the second operand (maximum length: 16 bits) will be valid only from 0-15.

Zero specifies a 1-bit length, and 15 specifies a 16-bit length. Bit field data may overlap the byte boundary of memory.

Note: For correct operation, set the upper 4 bits of the 8-bit register used as the first operand to 0.

Bytes: 4

Example:

1. MOV DS0,IX,SRC,DWPTR ;Point to area to extract
2. MOV [IX],5555H ;Fill in sample patterns
3. MOV [IX+2],3333H
4. MOV CL,3 ;Start at bit 3

(A) MOV DL,4 ;(A)
(B) EXT CL,DL ;Extract 5 bits starting at 3 (B)
(C) EXT CL,12 ;Extract 13 bits starting at 8 (C)

at (A) memory =

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>0011</td>
<td>0011</td>
<td>0101</td>
</tr>
</tbody>
</table>

CL = 3, IX = base, AW = unknown

at (B)

CL = 8, IX = base, AW = (0000 0000 000)01010

at (C)

CL = 5, IX = base + 2, AW = (000)1 0011 0101 0101
INPUT/OUTPUT

IN acc,imm8
Input specified I/O device

<table>
<thead>
<tr>
<th>W</th>
<th>imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 1 1 0 0 1 0</td>
</tr>
</tbody>
</table>

When W=0 AL ← (imm8)
When W=1 AH ← (imm8+1), AL ← (imm8)
Inputs the contents of the I/O device specified by the second operand to the accumulator (AL or AH) specified by the first operand.

Bytes: 2
Clock:
  When W=0: 9
  When W=1: 13, µPD70108
  13, µPD70116 odd addresses
  9, µPD70116 even addresses

Transfers: 1
Flag operation: None
Example: IN AL,20H

IN acc,DW
Input to device indirectly specified by DW

<table>
<thead>
<tr>
<th>W</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 1 1 0 1 1 0</td>
</tr>
</tbody>
</table>

When W=0: AL ← (DW)
When W=1: AH ← (DW+1),AL ← (DW)
Inputs the contents of the I/O device specified by the DW register to the accumulator (AL or AW) specified by the first operand.

Bytes: 1
Clocks:
  When W=0: 8
  When W=1: 12, µPD70108
  12, µPD70116 odd addresses
  8, µPD70116 even addresses

Transfers: 1
Flag Operation: None
Example: IN AL,DW
OUT imm8,acc
Output to directly specified I/O device

<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

When W=0: (imm8) ← AL
When W=1: (imm8+1) ← AH, (imm8) ← AL

Outputs the contents of the accumulator (AL or AH) specified by the second operand to the I/O device specified by the first operand.

Bytes: 2
Clocks: 8
Transfers: 1
Flag operation: None
Example: OUT 30H,AW

OUT DW,acc
Output to indirectly specified (by DW) I/O device

<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

When W=0: (DW) ← AL
When W=1: (DW+1) ← AH, (DW) ← AL

Outputs the contents of the accumulator (AL or AW) specified by the second operand to the I/O device specified by the first operand.

Bytes: 1
Clocks: 8, μPD70108
12, μPD70116 odd addresses
8, μPD70116 even addresses
Transfers: 1
Flag operation: None
Example: OUT DW,AW
OUT DW,acc
Output to indirectly specified (by DW) I/O device

| 7 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | W |

When W=0: (DW) ← AL
When W=1: (DW+1) ← AH, (DW) ← AL

Outputs the contents of the accumulator (AL or AW) specified by the second operand to the I/O device specified by the first operand.

Bytes: 1

Clocks:
- When W=0: 8
- When W=1: 12, μPD70108
  12, μPD70116 odd addresses
  8, μPD70116 even addresses

Transfers: 1

Flag operation: None

Example: OUT DW,AW
**PRIMITIVE INPUT/OUTPUT**

(Repeat) INM [DS1-spec:]dst-block,DW

Input multiple

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
</table>

When \( W=0 \): (\( IY \) ← (DW))
- \( \text{Dir}=0 \): \( IY ← IY+1 \)
- \( \text{Dir}=1 \): \( IY ← IY-1 \)

When \( W=1 \): (\( IY+1, IY \) ← (DW+1,DW))
- \( \text{Dir}=0 \): \( IY ← IY+2 \)
- \( \text{Dir}=1 \): \( IY ← IY-2 \)

Transfers the contents of the I/O device addressed by the DW register to the memory location addressed by the \( IY \) index register.

When this instruction is paired with a repeat prefix (REP), the REP prefix controls the number of times the transfer will be repeated. When transfers are repeated, the contents (address of the I/O device) of the DW register are fixed. However, to transfer the next byte or word, the IX index register is automatically incremented (+1 or +2) or decremented (−1 or −2) each time one byte or word is transferred. The direction of the block is determined by the direction flag (DIR).

Byte or word specification is performed according to the attribute of the operand. The destination block must always be located within the segment specified by the DS1 segment register, and a segment override prefix is prohibited.

**Bytes:** 1

**Clocks:**

**Repeat:**
- When \( W=0 \): \( 9+8/\text{rep} \)
- When \( W=1 \): \( 9+16/\text{rep}: \mu PD70108 \)
  - \( 9+16/\text{rep}: \mu PD70116 \) odd-odd addresses
  - \( 9+12/\text{rep}: \mu PD70116 \) odd-even addresses
  - \( 9+8/\text{rep}: \mu PD70116 \) even-even addresses

**Single operation:**
- When \( W=0 \): 10
- When \( W=1 \): 18, \( \mu PD70108 \)
  - 18, \( \mu PD70116 \) odd-odd addresses
  - 14, \( \mu PD70116 \) odd-even addresses
  - 10, \( \mu PD70116 \) even-even addresses

**Transfers:**
- Repeat: \( 2/\text{rep} \)
- Single operation: 2

**Flag operation:** None

**Example:**

```asm
MOV CW,30
MOV IY,OFFSET BYTE_VAR
REP INM BYTE_VAR,DW
; Input 30 bytes
```
OUTM DW,[seg-spec:src-block]

Output multiple

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

When \( W=0 \):
\[ (DW) \leftarrow (IX) \]
\[ \text{DIR}=0: \ IX \leftarrow IX+1 \]
\[ \text{DIR}=1: \ IX \leftarrow IX-1 \]

When \( W=1 \):
\[ (DW+1, DW) \leftarrow (IX+1, IX) \]
\[ \text{DIR}=0: \ IX \leftarrow IX+2 \]
\[ \text{DIR}=1: \ IX \leftarrow IX-2 \]

Transfers the memory contents addressed by the IX index register to the I/O device addressed by the DW register. When this instruction is paired with a repeat prefix (REP), REP controls the number of times the transfer will be repeated. When transfers are repeated, the contents (address of the I/O device) of the DW register are fixed. However, to transfer the next byte or word, the IX index register is automatically incremented (+1 or +2) or decremented (-1 or -2) each time one byte or word is transferred. The direction or the block is determined by the direction flag (DIR).

Byte or word specification is performed according to the attribute of the operand. The default segment register for the source block is DS0, and segment override is possible. The source block may be located within the segment specified by any (optional) segment register.

Bytes: 1
Clocks: 2
Repeat:
When \( W=0 \): \( 9+8/\text{rep} \)
When \( W=1 \): \( 9+16/\text{rep}, \muPD70108 \)
\[ 9+16/\text{rep}, \muPD70116 \text{ odd-odd addresses} \]
\[ 9+12/\text{rep}, \muPD70116 \text{ odd-even addresses} \]
\[ 9+8/\text{rep}, \muPD70116 \text{ even-even addresses} \]

Single operation:
When \( W=0 \): 10
When \( W=1 \): 18, \( \muPD70108 \)
\[ 18, \muPD70116 \text{ odd-odd addresses} \]
\[ 14, \muPD70116 \text{ odd-even addresses} \]
\[ 10, \muPD70116 \text{ even-even addresses} \]

Transfers:
Repeat: \( 2/\text{rep} \)
Single operation: 2
Flag operation: None
Example:
\[ \text{REP OUTM DW,BYTE PTR DS1:}[IX] \]
ADD mem, reg
Add memory with register to memory

- **7**
  - 0 0 0 0 0 0 0 0
  - W
- **mod**
  - **reg**
  - **mem**

(mem) ← (mem) + reg

Add the contents of the 8- or 16-bit register specified by the second operand to the 8- or 16-bit memory contents addressed by the first operand. Stores the result in the memory location addressed by the first operand.

**Bytes:** 2/3/4

**Clocks:**
- When W=0: 16
- When W=1: 24, µPD70108
  - 24, µPD70116 odd addresses
  - 16, µPD70116 even addresses

**Transfers:** 2

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

- ADD WORD_VAR, AW
- ADD [IX], CW

ADD reg, mem
Add register with memory to register

- **7**
  - 0 0 0 0 0 0 0 1
  - W
- **mod**
  - **reg**
  - **mem**

reg ← reg + (mem)

Add the 8- or 16-bit memory contents addressed by the second operand to the contents of the 8- or 16-bit register specified by the first operand. Stores the result in the register specified by the first operand.

**Bytes:** 2/3/4

**Clocks:**
- When W=0: 11
- When W=1: 15, µPD70108
  - 15, µPD70116 odd addresses
  - 11, µPD70116 even addresses

**Transfers:** 1

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

- ADD AW, WORD_VAR
- ADD BW, [BP][IX]
ADD reg,imm

Add register with immediate data to register

```
    7 6 5 4 3 2 1 0
  1 0 0 0 0 0 0 S W
```

```
  1 1 0 0 0 0 reg
```

```
<table>
<thead>
<tr>
<th>imm8 or imm16-low</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm16-high</td>
</tr>
</tbody>
</table>
```

reg ← reg + imm

Adds the 8- or 16-bit immediate data specified by the second operand to the contents of the 8- or 16-bit register specified by the first operand, and stores the result in the register specified by the first operand.

Bytes: 2/3/4
Clocks: 4
Transfers: None
Flag operation:

```
<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
```

Example: ADD DL,10

ADD mem,imm

Add memory with immediate data to memory

```
    7 6 5 4 3 2 1 0
  1 0 0 0 0 0 0 S W
```

```
<table>
<thead>
<tr>
<th>mod 0 0 0 0 mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>(disp-low)</td>
</tr>
<tr>
<td>(disp-high)</td>
</tr>
<tr>
<td>imm8 or imm16-low</td>
</tr>
<tr>
<td>imm16-high</td>
</tr>
</tbody>
</table>
```

(mem) ← (mem) + imm

Adds the 8- or 16-bit immediate data specified by the second operand to the 8- or 16-bit memory contents addressed by the first operand. Stores the result in the memory location addressed by the first operand.

Bytes: 3/4/5/6
Clocks:

- When W=0: 18
- When W=1: 26, μPD70108
  26, μPD70116 odd addresses
  18, μPD70116 even addresses
Transfers: 2
Flag operation:

```
<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
```

Example:

ADD BYTE_VAR[BP],100
ADD WORD_VAR[BW][IX],1234H
ADD acc,imm

Add accumulator with immediate data to accumulator

When W=0:  AL ← AL imm
When W=1:  AW ← AW imm

Adds the 8- or 16-bit immediate data specified by the second operand to the contents of the accumulator (AL or AW) specified by the first operand. Stores the result in the accumulator specified by the first operand.

Bytes: 2/3
Clocks: 4
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

ADD AL,3
ADD AW,2000H

ADDC reg,reg

Add with carry, register with register to register

reg ← reg + reg + CY

Advis the contents of the 8- or 16-bit register specified by the second operand and the contents of the carry flag to the contents of the 8- or 16-bit register specified by the first operand. Stores the result in the register specified by the first operand.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation:

Example: ADDC BW,DW
ADD C mem, reg
Add with carry, memory with register to memory

(mem) ← (mem) + reg + CY
Adds the contents of the 8- or 16-bit register specified by the second operand and the contents of the carry flag to the 8- or 16-bit memory contents addressed by the first operand. Stores the result in the memory location addressed by the first operand.

Bytes: 2/3/4
Clocks:
When \( W=0 \): 16
When \( W=1 \): 24, \( \mu \)PD70108
24, \( \mu \)PD70116 odd addresses
16, \( \mu \)PD70116 even addresses

Transfers: 2
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: ADDC WORD_VAR, CW

ADD C reg, mem
Add with carry, register with memory to register

reg ← reg + (mem) + CY
Adds the 8- or 16-bit memory contents addressed by the second operand and the contents of the carry flag to the contents of the 8- or 16-bit register specified by the first operand. Stores the result in the register specified by the first operand.

Byte: 2/3/4
Clocks:
When \( W=0 \): 11
When \( W=1 \): 15, \( \mu \)PD70108
15, \( \mu \)PD70116 odd addresses
11, \( \mu \)PD70116 even addresses

Transfers: 1
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Examples:
ADD C AW, WORD_VAR
ADD C BW, [BP][IX]
ADD reg,imm
Add with carry, register with immediate data to register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm8 or imm16-low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm16-high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

reg ← reg + imm + CY
Adds the 8- or 16-bit immediate data specified by the second operand and the contents of the carry flag to the contents of the 8- or 16-bit register specified by the first operand. Stores the result in the register specified by the first operand.

Bytes: 3/4
Clocks: 4
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
ADDCC CW,404H
ADDCC DL,3

ADD mem,imm
Add with carry, memory with immediate data to memory

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>W</td>
</tr>
<tr>
<td>mod</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp-low)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp-high)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm8 or imm16-low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm16-high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(mem) ← (mem) + imm + CY
Adds the 8- or 16-bit immediate data specified by the second operand and the contents of the carry flag to the 8- or 16-bit memory contents addressed by the first operand. Stores the result in the memory location addressed by the first operand.

Bytes: 3/4/5/6
Clocks:
When W=0: 18
When W=1: 26, µPD70108
26, µPD70116 odd addresses
18, µPD70116 even addresses
Transfers: 2
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: ADDCC WORD_VAR,2000H
ADD acc,imm
Add with carry, accumulator with immediate data to accumulator

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

imm8 or imm16-low

| 1 | 1 |

imm16-high

When W=0: AL ← AL + imm8 + CY
When W=1: AW ← AW + imm16 + CY

Adds the 8- or 16-bit immediate data specified by the second operand and the contents of the carry flag to the accumulator (AL or AW) specified by the first operand. Stores the result in the accumulator specified by the first operand.

Bytes: 2/3
Clocks: 4
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: ADDC AL,7

SUB reg,reg
Subtract register from register to register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

reg ← reg - reg

Subtracts the contents of the 8- or 16-bit register specified by the second operand from the contents of the 8- or 16-bit register specified by the first operand. Stores the result in the register specified by the first operand.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: SUB BW,CW
**SUB mem,reg**

Subtract register from memory to memory

\[
\begin{array}{ccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & W \\
\end{array}
\]

\[
\begin{array}{c}
\text{mod} \\
\text{reg} \\
\text{mem} \\
\end{array}
\]

\[(\text{mem}) - (\text{mem}) - \text{reg}\]

Subtracts the contents of the 8- or 16-bit register specified by the second operand from the 8- or 16-bit memory contents addressed by the first operand. Stores the result in the memory location addressed by the first operand.

**Bytes:** 2/3/4

**Clocks:**
- When \( W=0 \): 16
- When \( W=1 \): 24, \( \mu \text{PD70108} \)
  - 24, \( \mu \text{PD70116} \) odd addresses
  - 16, \( \mu \text{PD70116} \) even addresses

**Transfers:** 2

**Flag operation:**

\[
\begin{array}{ccccccc}
V & S & Z & AC & P & CY \\
X & X & X & X & X & X \\
\end{array}
\]

**Example:**

\[
\begin{array}{c}
\text{SUB} \\
\text{WORD} \text{VAR,BW} \\
\text{SUB} \\
[\text{IX}] \text{AL} \\
\end{array}
\]

---

**SUB reg,mem**

Subtract memory from register to register

\[
\begin{array}{ccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & W \\
\end{array}
\]

\[
\begin{array}{c}
\text{mod} \\
\text{reg} \\
\text{mem} \\
\end{array}
\]

\[(\text{disp-low}) \\
(\text{disp-high})\]

\[(\text{mem}) - \text{reg}\]

Subtracts the 8- or 16-bit memory contents addressed by the second operand from the 8- or 16-bit register specified by the first operand. Stores the result in the register specified by the first operand.

**Bytes:** 2/3/4

**Clocks:**
- When \( W=0 \): 11
- When \( W=1 \): 15, \( \mu \text{PD70108} \)
  - 15, \( \mu \text{PD70116} \) odd addresses
  - 11, \( \mu \text{PD70116} \) even addresses

**Transfers:** 1

**Flag operation:**

\[
\begin{array}{ccccccc}
V & S & Z & AC & P & CY \\
X & X & X & X & X & X \\
\end{array}
\]

**Example:** SUB CW,WORD\_VAR
**SUB reg,imm**

Subtract immediate from register to register

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>W</td>
</tr>
</tbody>
</table>

1 1 1 0 1 0 1  reg

imm8 or imm16-low

imm16-high
```

\[ \text{reg} \leftarrow \text{reg} - \text{imm} \]

Subtracts the 8- or 16-bit immediate data specified by the second operand from the contents of the 8- or 16-bit register specified by the first operand. Stores the result in the register specified by the first operand.

Bytes: 3/4

Clocks: 4

Transfers: None

Flag operation:

```
V S Z AC P CY
X X X X X X
```

Example: SUB IX,4

---

**SUB mem,imm**

Subtract immediate data from memory to memory

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>W</td>
</tr>
</tbody>
</table>

1 0 1 0 1 1 1 1 mem

mod 1 0 1 mem

(disp-low)

(disp-high)

imm8 or imm16-low

imm16-high
```

\[ \text{(mem)} \leftarrow \text{(mem)} - \text{imm} \]

Subtracts the 8- or 16-bit immediate data specified by the second operand from the 8- or 16-bit memory contents addressed by the first operand. Stores the result in the memory location addressed by the first operand.

Bytes: 3/4/5/6

Clocks:

- When \( W = 0 \): 18
- When \( W = 1 \): 26, \( \mu \text{PD70108} \)
  26, \( \mu \text{PD70116} \) odd addresses
  18, \( \mu \text{PD70116} \) even addresses

Transfers: 2

Flag operation:

```
V S Z AC P CY
X X X X X X
```

Example: SUB WORD_VAR,10
SUB acc,imm
Subtract immediate data from accumulator to accumulator

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

imm8 or imm16-low

imm16-high

When W=0: AL ← AL − imm8
When W=1: AW ← AW − imm16

Subtracts the 8- or 16-bit immediate data specified by the second operand from the accumulator (AL or AW) specified by the first operand. Stores the result in the accumulator specified by the first operand.

Bytes: 2/3
Clocks: 4
Transfers: None
Flag operation:

Example: SUB AL,8

SUBC reg,reg
Subtract with carry, register from register to register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td>reg</td>
</tr>
</tbody>
</table>

reg ← reg − reg − CY

Subtracts the contents of the 8- or 16-bit register specified by the second operand and the contents of the carry flag from the 8- or 16-bit register specified by the first operand.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation:

Example: SUBC BW,DW
### SUBC mem, reg

Subtract with carry, register from memory to memory

\[
\begin{array}{cccccccc}
7 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & W \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{mod} & \text{reg} & \text{mem} \\
\text{(disp-low)} & \text{(disp-high)} \\
\end{array}
\]

\[(\text{mem}) \leftarrow (\text{mem}) - \text{reg} - \text{CY}\]

Subtracts the contents of the 8- or 16-bit register specified by the second operand and the contents of the carry flag from the 8- or 16-bit memory contents specified by the first operand. Stores the result in the memory location addressed by the first operand.

Bytes: 2/3/4

Clocks:

- When \(W=0\): 16
- When \(W=1\): 24, \(\mu PD70108\)
  - 24, \(\mu PD70116\) odd addresses
  - 16, \(\mu PD70116\) even addresses

Transfers: 2

Flag operation:

\[
\begin{array}{cccccccc}
V & S & Z & AC & P & CY \\
X & X & X & X & X & X \\
\end{array}
\]

Example: \text{SUBC} BYTE\_VAR, AL

### SUBC reg, mem

Subtract with carry, memory from register to register

\[
\begin{array}{cccccccc}
7 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & W \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{mod} & \text{reg} & \text{mem} \\
\text{(disp-low)} & \text{(disp-high)} \\
\end{array}
\]

\[\text{reg} \leftarrow \text{reg} - (\text{mem}) - \text{CY}\]

Subtracts the contents of the 8- or 16-bit memory addressed by the second operand and the contents of the carry flag from the 8- or 16-bit register specified by the first operand. Stores the result in the register specified by the first operand.

Bytes: 2/3/4

Clocks:

- When \(W=0\): 11
- When \(W=1\): 15, \(\mu PD70108\)
  - 15, \(\mu PD70116\) odd addresses
  - 11, \(\mu PD70116\) even addresses

Transfers: 1

Flag operation:

\[
\begin{array}{cccccccc}
V & S & Z & AC & P & CY \\
X & X & X & X & X & X \\
\end{array}
\]

Example: \text{SUBC} AW,WORD\_VAR
**SUBC reg,imm**

Subtract with carry, immediate data from register to register:

```
0 7
| 1 | 0 | 0 | 0 | 0 | 0 | S | W |
```

```
| 1 | 1 | 0 | 1 | 1 | reg |
```

```
imm8 or imm16-low
```

```
imm16-high
```

reg ← reg – imm – CY

Subtracts the contents of the 8- or 16-bit immediate data specified by the second operand and the contents of the carry flag from the 8- or 16-bit register specified by the first operand. Stores the result in the register specified by the first operand.

Bytes: 3/4

Clocks: 4

Transfers: None

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: SUBC DL,10

**SUBC mem,imm**

Subtract with carry, immediate data from memory to memory:

```
0 7
| 1 | 0 | 0 | 0 | 0 | 0 | S | W |
```

```
mod 0 1 1 mem
```

```
(disp-low)
```

```
(disp-high)
```

```
imm8 or imm16-low
```

```
imm16-high
```

(mem) ← (mem) – imm – CY

Subtracts the contents of the 8- or 16-bit immediate data specified by the second operand and the contents of the carry flag from the 8- or 16-bit memory contents addressed by the first operand. Stores the result in the memory location addressed by the first operand.

Bytes: 3/4/5/6

Clocks:

- When W=0: 18
- When W=1: 26, µPD70108
  - 26, µPD70116 odd addresses
  - 18, µPD70116 even addresses

Transfers: 2

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: SUBC WORD_VAR,25
SUBC acc,imm

Subtract with carry, immediate data from accumulator to accumulator

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>W</td>
</tr>
</tbody>
</table>

- **imm** or imm16-low
- **imm16-high**

When W=0: \( AL \leftarrow AL - \text{imm8} - CY \)
When W=1: \( AW \leftarrow AW - \text{imm16} - CY \)

Subtracts the 8- or 16-bit immediate data specified by the second operand and the contents of the carry flag from the accumulator (AL or AW) specified by the first operand. Stores the result in the accumulator specified by the first operand.

Bytes: 2/3
Clocks: 4
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: SUBC AL,8
BCD ARITHMETIC

ADD4S [DS1-spec:]dst-string,[seg-spec:]src-string
ADD4S (no operand)

Add nibble string

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

BCD string (IY,CL) ← BCD string (IY,CL) + BCD string (IX,CL)

Adds the packed BCD string addressed by the IX index register to the packed BCD string addressed by the IY index register. Stores the result in the string addressed by the IY register. The length of the string (number of BCD digits) is specified by the CL register and can vary from 1 to 254 digits.

When the number of digits is even, the zero and carry flags will be set according to the result of the operation. When the number of digits is odd, the zero and carry flags may not be set correctly. In this case, (CL = odd), the zero flag will not be set unless the upper 4 bits of the highest byte are all zero. The carry flag will not be set unless there is a carry out of the upper 4 bits of the highest byte. When CL is odd, the contents of the upper 4 bits of the highest byte of the result are undefined.

The destination string must always be located within the segment specified by the DS1 segment register. Segment override is prohibited.

The default segment register for the source string is DS0 and segment override is possible. The source string may be located within the segment specified by any (optional) segment register.

The format for the packed BCD string follows.

Bytes: 2
Clocks: 7 + 19n, where n = one-half the number of BCD digits
Transfers: 3n
Flag operation:

Example: See example for CMP4S
**Section 12**

**Instruction Set**

**SUB4S [DS1-spec:]dst-string,[seg-spec:]src-string**

SUB4S (no operand)

Subtract nibble string

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

BCD string (IY,CL) ← BCD string (IY,CL) − BCD string (IX,CL)

Subtracts the packed BCD string addressed by the IX index register from the packed BCD string addressed by the IY index register. Stores the result in the string addressed by the IY register.

The length of the string (number of BCD digits) is specified by the CL register and can vary from 1 to 254 digits.

When the number of digits is even, the zero and carry flags will be set according to the result of the operation. When the number of digits is odd, the zero and carry flags may not be set correctly. In this case, (CL = odd), the zero flag will not be set unless the upper 4 bits of the highest byte are all zero. The carry flag will not be set unless there is a carry out of the upper 4 bits of the highest byte. When CL is odd, the contents of the upper 4 bits of the highest byte of the result are undefined.

The destination string must always be located within the segment specified by the DS₁ segment register. Segment override is prohibited.

The default segment register for the source string is DS₀, and segment override is possible. The source string may be located within the segment specified by any (optional) segment register.

The format for the packed BCD string is shown as follows.

Bytes: 2

Clocks: \(7 + 19n\), where \(n = \text{one-half the number of BCD digits}\)

Transfers: \(3n\)

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>X</td>
<td>U</td>
<td>U</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: See example for CMP4S
**CMP4S**

[DS1-spec:]dst-string,[seg-spec:]src-string

CMP4S (no operand)

Compare nibble string

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 0 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

BCD string (IY,CL) — BCD string (IX,CL)

Subtracts the packed BCD string addressed by the IX index register from the packed BCD string addressed by the IY index register. The result is not stored and only the flags are affected. The length of the string (number of BCD digits) is specified by the CL register and can vary from 1 to 254 digits.

When the number of digits is even, the zero and carry flags will be set according to the result of the operation. When the number of digits is odd, the zero and carry flags may not be set correctly. In this case, (CL = odd), the zero flag will not be set unless the upper 4 bits of the highest byte are all zero. The carry flag will not be set unless there is a carry out of the upper 4 bits of the highest byte. When CL is odd, the contents of the upper 4 bits of the highest byte of the result are undefined.

The default segment register for the source string is DS0 and segment override is possible.

The source string may be located within the segment specified by any (optional) segment register. The format for the packed BCD string is shown below.

Bytes: 2

Clocks: 7 + 19n, where n = one-half the number of BCD digits

Transfers: 2

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>X</td>
<td>U</td>
<td>U</td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

µPD70116 BCD string operation

MOV AW,PS ;Set both data segments to
MOV DS0,AW ;same as program
MOV DS1,AW ;segment
MOV IX,OFFSET STR0 ;Point to BCD strings
MOV IY,OFFSET STR1
MOV CL,8 ;Eight digits
;in strings (A)
CMP4S ;Compare (B)
ADD4S ;Add string0
;to string1 (C)
CMP4S ;Compare again (D)
SUB4S ;Subtract string0
;from string1 (E)
SUB4S ;again (result is zero) (F)
SUB4S ;and again (underflow) (G)
HALT

; STR0 DW 4321H,0765H ;BCD# 07654321
; STR1 DW 4321H,0765H ;BCD# 07654321
;
; at (A), STR0 = 7654321,
; STR1 = 7654321, Z = ?, CY = ?
; at (B), STR0 = 7654321,
; STR1 = 7654321, Z = 1, CY = 0
; at (C), STR0 = 7654321,
; STR1 = 15308642, Z = 0, CY = 0
; at (D), STR0 = 7654321,
; STR1 = 15308642, Z = 0, CY = 0
; at (E), STR0 = 7654321,
; STR1 = 7654321, Z = 0, CY = 0
; at (F), STR0 = 7654321,
; STR1 = 00000000, Z = 1, CY = 0
; at (G), STR0 = 7654321,
; STR1 = 92345679, Z = 0, CY = 1
;

![Memory Diagram](image-url)
ROL4 reg8
Rotate left nibble, 8-bit register

Treats the byte data of the 8-bit register specified by the operand as a two-digit BCD and uses the lower 4 bits of the AL register (AL_L) to rotate that data one digit to the left.

Due to the result of this instruction, the contents of the upper 4 bits of the AL register are not assured.

Bytes: 3
Clocks: 25
Transfers: None
Flag operation: None
Example:
MOV BL,95H
MOV AL,03H
ROL4 BL ;BL = 53H, AL = X9H

ROL4 mem8
Rotate left nibble, 8-bit memory

Treats the byte data of the 8-bit memory location addressed by the operand as a two-digit BCD and uses the lower 4 bits of the AL register (AL_L) to rotate that data one digit to the left.

Due to the result of this instruction, the contents of the upper 4 bits of the AL register are not assured.

Bytes: 3/4/5
Clocks: 28
Transfers: 2
Flag operation: None
Example:
MOV BYTE PTR [IX],12H
MOV AL,03H
ROL4 [IX] ;[IX] = 23H, AL = X1H
ROR4 reg8
Rotate right nibble, 8-bit register

Treats the byte data of the 8-bit register specified by the operand as two-digit BCD and uses the lower 4 bits of the AL register (AL_L) to rotate the data one digit to the right.

Due to the result of this instruction, the contents of the upper 4 bits of the AL register are not assured.

Bytes: 3
Clocks: 29
Transfers: None
Flag operation: None

Example:
MOV CL,95H
MOV AL,03H
ROR4 CL ;CL = 39H, AL = X5H

ROR4 mem8
Rotate right nibble, 8-bit memory

Treats the byte data of the 8-bit memory location addressed by the operand as a two-digit BCD and uses the lower 4 bits of the AL register (AL_L) to rotate that data one digit to the right. Due to the result of this instruction, the contents of the upper 4 bits of the AL register are not assured.

Bytes: 3/4/5
Clocks: 33
Transfers: 2
Flag operation: None

Example:
MOV BYTE PTR [IX],12H
MOV AL,03H
ROR4 [IX] ;[IX] = 31H, AL = X2H
INC reg8
Increment 8-bit register

\[
\begin{array}{c|cccccccc}
7 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
\hline
1 & 1 & 0 & 0 & 0 & \text{reg}
\end{array}
\]

reg8 \leftarrow reg + 1

Increments by 1 the contents of the 8-bit register specified by the operand.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: INC BL

INC mem
Increment memory

\[
\begin{array}{cccccccc}
7 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
\hline
1 & 1 & \text{mod} & 0 & 0 & 0 & \text{mem} & W \\
\hline
\text{(disp-low)} & \text{(disp-high)}
\end{array}
\]

(mem) \leftarrow (mem) + 1

Increments by 1 the contents of the 8- or 16-bit memory location specified by the operand.

Bytes: 2/3/4
Clocks:
- When W=0: 16
- When W=1: 24, \( \mu \text{PD70108} \)
  - 24, \( \mu \text{PD70116} \) odd addresses
  - 16, \( \mu \text{PD70116} \) even addresses
Transfers: 2
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

INC WORD_VAR
INC BYTE PTR [BW]
INC reg16

Increment 16-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

reg16 — reg16 + 1

Increments by 1 the contents of the 16-bit register specified by the operand.

Bytes : 1

Clocks: 2

Transfers: None

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

INC BW
INC IX

DEC reg8

Decrement 8-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

reg8 — reg8 — 1

Decrements by 1 the contents of the 8-bit register specified by the operand.

Bytes: 2

Clocks: 2

Transfers: None

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: DEC DH
DEC mem

Decrement memory

\[
\begin{array}{cccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & W \\
\end{array}
\]

\[
\begin{array}{c}
\text{mod} \ 0 \ 0 \ 1 \ \text{mem} \\
\end{array}
\]

\[
\begin{array}{c}
\text{(disp-low)} \\
\end{array}
\]

\[
\begin{array}{c}
\text{(disp-high)} \\
\end{array}
\]

\[(\text{mem}) \leftarrow (\text{mem}) - 1\]

Decrement by 1 the 8- or 16-bit memory contents addressed by the operand.

Bytes: 2/3/4

Clocks:

- When W=0: 16
- When W=1: 24, \(\mu\)PD70108
- 24, \(\mu\)PD70116 odd addresses
- 16, \(\mu\)PD70116 even addresses

Transfers: 2

Flag operation:

\[
\begin{array}{cccccccc}
\text{V} & \text{S} & \text{Z} & \text{AC} & \text{P} & \text{CY} \\
\text{X} & \text{X} & \text{X} & \text{X} & \text{X} & \text{X} \\
\end{array}
\]

Example:

- DEC BYTE_VAR
- DEC WORD_VAR[BW][IX]

DEC reg16

Decrement 16-bit register

\[
\begin{array}{cccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & \text{reg} \\
\end{array}
\]

\[\text{reg16} \leftarrow \text{reg16} - 1\]

Decrement by 1 the contents of the 16-bit register specified by the operand.

Bytes: 1

Clocks: 2

Transfers: None

Flag operation:

\[
\begin{array}{cccccccc}
\text{V} & \text{S} & \text{Z} & \text{AC} & \text{P} & \text{CY} \\
\text{X} & \text{X} & \text{X} & \text{X} & \text{X} & \text{X} \\
\end{array}
\]

Example: DEC BP
MULTIPLICATION

MULU reg8

Multiply unsigned, 8-bit register

\[
\begin{array}{cccccc}
7 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & & \text{reg}
\end{array}
\]

\[\text{AW} \leftarrow \text{AL} \times \text{reg8}\]

When \(\text{AH}=0\): \(\text{CY} \leftarrow 0, \text{V} \leftarrow 0\)
When \(\text{AH} \neq 0\): \(\text{CY} \leftarrow 1, \text{V} \leftarrow 1\)

Performs unsigned multiplication of the contents of the AL register and the contents of the 8-bit register specified by the operand. Stores the word result in the AL and AH registers. When the upper half (AH) of the result is not 0, the carry and overflow flags are set.

Bytes: 2

Clocks: 21 or 22 (according to data)

Transfers: None

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

\[
\begin{array}{l}
\text{MOV AL,13} ; \text{AW} = \text{XX0DH} \\
\text{MOV CL,5} \\
\text{MULU CL} ; \text{AW} = 0041H = 65, C = V = 0
\end{array}
\]

MULU mem8

Multiply unsigned, 8-bit memory

\[
\begin{array}{cccccc}
7 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & & \text{mem}
\end{array}
\]

\[\text{mod} \times 1 \ 0 \ 0 \ \text{disp-low}
\]

\[\text{disp-high}\]

\[\text{AW} \leftarrow \text{AL} \times (\text{mem8})\]

When \(\text{AH}=0\): \(\text{CY} \leftarrow 0, \text{V} \leftarrow 0\)
When \(\text{AH} \neq 0\): \(\text{CY} \leftarrow 1, \text{V} \leftarrow 1\)

Performs unsigned multiplication of the contents of the AL register and the 8-bit memory location addressed by the operand. Stores the word result in the AL and AH registers. When the upper half (AH) of the result is not 0, the carry and overflow flags are set.

Bytes: 2/3/4

Clocks: 27 or 28 (according to data)

Transfers: 1

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

\[
\begin{array}{l}
\text{MOV AL,35} \\
\quad ; \text{AW} = \text{XX23H} \\
\text{MOV BYTE_VAR,20} \\
\text{MULU BYTE_VAR} ; \text{AW} = 02BCH = 700, C = V = 1
\end{array}
\]

\[\bullet\]

\[\bullet\]

\[\text{MULU BYTE PTR [IX]}\]
MULU reg16
Multiply unsigned, 16-bit register

```
  7 6 5 4 3 2 1 0
0   1 1 1 1 0 1 1 1
1   1 1 1 0 0 0
```

DW, AW ← AW × reg16
When DW=0: CY ← 0, V ← 0
When DW≠0: CY ← 1, V ← 1

Performs unsigned multiplication of the contents of the AW register and the contents of the 16-bit register specified by the operand. Stores the double-word result in the AW and DW registers. When the upper half (DW) of the result is not 0, the carry and overflow flags are set.

Bytes: 2
Clocks: 29 or 30 (according to data)
Transfers: None
Flag operation:

```
V S Z AC P CY
X U U U U X
```

Example:
MOV AW,1234H
MOV CW,3
MULU CW
;DW = 0000H, AW = 369CH, ;C = V = 0

MULU mem16
Multiply unsigned, 16-bit memory

```
  7 6 5 4 3 2 1 0
0   1 1 1 1 0 1 1 1
```

mod 1 0 0 mem

```
(disp-low)
(disp-high)
```

DW, AW ← AW × (mem16)
When DW=0: CY ← 0, V ← 0
When DW≠0: CY ← 1, V ← 1

Performs unsigned multiplication of the contents of the AW register and the 16-bit memory contents addressed by the operand. Stores the double-word result in the AW and DW registers. When the upper half (DW) of the result is not 0, the carry and overflow flags are set.

Bytes: 2/3/4
Clocks: 39 or 40, µPD70108
39 or 40, µPD70116 odd addresses
35 or 36, µPD70116 even addresses
Transfers: 1
Flag operation:

```
V S Z AC P CY
X U U U U X
```

Example:
MOV AW,400H
MOV WORD_VAR,9310H
MULU WORD_VAR
;DW = 024CH, AW = 4000H, ;C = V = 1
**MUL reg8**

Multiply signed, 8-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ AW \leftarrow AL \times \text{reg8} \]

When AH=sign extension of AL: CY \leftarrow 0, V \leftarrow 0

When AH=sign extension of AH: CY \leftarrow 1, V \leftarrow 1

Performs signed multiplication of the contents of the AL register and the contents of the 8-bit register specified by the operand. Stores the double-word result in the AL and AH registers. When the upper half (AH) of the result is not the sign extension of the lower half (AL), the carry and overflow flags are set.

Bytes: 2

Clocks: 33 to 39 (according to data)

Transfers: None

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

<table>
<thead>
<tr>
<th>MOV</th>
<th>AL, 18</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>;AW = XX12H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MOV</th>
<th>CL, -2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>;CL = FEH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MUL</th>
<th>CL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>;AW = FFDC = -36, C = V = 0</td>
</tr>
</tbody>
</table>

**MUL mem8**

Multiply signed, 8-bit memory

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ AW \leftarrow AL \times \text{(mem8)} \]

When AH=sign extension of AL: CY \leftarrow 0, V \leftarrow 0

When AH=sign extension of AH: CY \leftarrow 1, V \leftarrow 1

Performs signed multiplication of the contents of the AL register and the 8-bit memory location addressed by the operand. Stores the double-word result in the AL and AH registers. When the upper half (AH) of the result is not the sign extension of the lower half (AL), the carry and overflow flags are set.

Bytes: 2/3/4

Clocks: 39 to 45 (according to data)

Transfers: None

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

<table>
<thead>
<tr>
<th>MOV</th>
<th>AL, 100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>;AW = XX64H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MOV</th>
<th>BYTE_VAR, -4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>; = FCH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MUL</th>
<th>BYTE_VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>;AW = FE70H = -400, C = V = 1</td>
</tr>
</tbody>
</table>
MUL reg16
Multiply signed, 16-bit register

\[
\begin{array}{cccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & & & & \text{reg}
\end{array}
\]

\[\text{DW, AW} \leftarrow \text{AW} \times \text{reg16}\]
When DW=sign extension of AW: CY ← 0, V ← 0
When DW≠sign extension of AH: CY ← 1, V ← 1

Performs signed multiplication of the contents of the AW register and the contents of the 16-bit register specified by the operand. Stores the double-word result in the AW and DW registers. When the upper half (DW) of the result is not the sign extension of the lower half (AW), the carry and overflow flags are set.

Bytes: 2
Clocks: 41 to 47 (according to data)
Transfers: None
Flag operation:

\[
\begin{array}{cccccccc}
V & S & Z & AC & P & CY \\
X & U & U & U & U & X
\end{array}
\]

Example:
MOV AW, -10
; AW = FFF6H
MOV BW, -10
; BW = FFF6H
MUL BW
; DW = 0000, AW = 0064H = 100,
; C = V = 0

MUL mem16
Multiply signed, 16-bit memory

\[
\begin{array}{cccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
& & & & & & & & \text{mem}
\end{array}
\]

\[\text{DW, AW} \leftarrow \text{AW} \times \text{mem16}\]
When DW=sign extension of AW: CY ← 0, V ← 0
When DW≠sign extension of AW: CY ← 1, V ← 1

Performs signed multiplication of the contents of the AW register and the 16-bit memory contents addressed by the operand. Stores the double-word result in the AW and DW registers. When the upper half (DW) of the result is not the sign extension of the lower half (AW), the carry and overflow flags are set.

Bytes: 2/3/4
Clocks:
51 to 57, μPD70108
51 to 57, μPD70116 odd addresses
47 to 53, μPD70116 even addresses
Transfers: 1
Flag operation:

\[
\begin{array}{cccccccc}
V & S & Z & AC & P & CY \\
X & U & U & U & U & X
\end{array}
\]

Example:
MOV AW, -10
; AW = FFF6H
MOV [IX], -20
; = FFEC
MUL WORD PTR [IX]
; DW = 0000, AW = 00C8H = 200,
; C = V = 0
MUL reg16,reg16,imm8
MUL reg16,imm8

Multiply signed, 16-bit register × 8-bit immediate data to 16-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

reg16 ← reg16 × imm8
Product ≤ 16 bits: CY ← 0, V ← 0
Product > 16 bits: CY ← 1, V ← 1

Performs signed multiplication of the contents of the 16-bit register specified by the second operand. (If a two-operand description, then performs signed multiplication on the contents specified by the first operand.) Performs signed multiplication on the 8-bit immediate data specified by the third operand. (If a two-operand description then performs signed multiplication on the data specified by the second operand.)

When the source register and the destination register are the same, a two-operand description is acceptable.

Bytes: 3
Clocks: 28 to 34 (according to data)
Transfers: None
Flag operation:

Example:
MUL AW,BW,10
;AW = BW*10
MUL CW,25
;CW = CW*25

MUL reg16,mem16,imm8

Multiply signed, 16-bit memory × 8-bit immediate data to 16-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

reg16 ← (MEM16) × imm8
Product ≤ 16 bits: CY ← 0, V ← 0
Product > 16 bits: CY ← 1, V ← 1

Performs signed multiplication of the contents of the 16-bit memory contents addressed by the second operand and the 8-bit immediate data specified by the third operand. Stores the result in the 16-bit register specified by the first operand.

Bytes: 3/4/5
Clocks:
38 to 44, μPD70108
38 to 44, μPD70116 odd addresses
34 to 40, μPD70116 even addresses
Transfers: 1
Flag operation:

Example:
MUL CW,WORD_VAR,7
;CW = [WORD_VAR]*7
MUL AW,[IX],22
;AW = [IX]*22
MUL reg16,reg16,imm16
MUL reg16,imm16
Multiply signed, 16-bit register × 16-bit immediate data to 16-bit register

\[
\begin{array}{cccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]

\begin{array}{cccc}
reg & reg & \\
\_ & \_ & \\
imm16-low & \\
imm16-high & \\
\end{array}

\[
\text{reg16} \leftarrow \text{reg16} \times \text{imm16}
\]

If product ≤ 16 bits: CY ← 0, V ← 0
If product > 16 bits: CY ← 1, V ← 1

Performs signed multiplication of the contents of the 16-bit register specified by the second operand — the first operand, when a two-operand description — and the 16-bit immediate data specified by the third (second) operand. Stores the result in the 16-bit register specified by the first operand.

When the source register and the destination register are the same, a two-operand description is possible.

Bytes: 4
Clocks: 36 to 42 (according to data)
Transfers: None
Flag operation:

\[
\begin{array}{cccccccc}
V & S & Z & AC & P & CY \\
X & U & U & U & U & X \\
\end{array}
\]

Example:
MUL AW,BW,200H
\; ;AW = BW*200H
MUL IX,300
\; ;IX = IX*300

MUL reg16,mem16,imm16
Multiply signed, 16-bit memory × 16-bit immediate data to 16-bit register

\[
\begin{array}{cccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]

\begin{array}{cccccc}
mod & reg & mem & \\
_ & _ & _ & _ & _ & \\
(disp-low) & \\
(disp-high) & \\
imm16-low & \\
imm16-high & \\
\end{array}

\[
\text{reg16} \leftarrow (\text{mem16}) \times \text{imm16}
\]

If product ≤ 16 bits: CY ← 0, V ← 0
If product > 16 bits: CY ← 1, V ← 1

Performs signed multiplication of the 16-bit memory contents specified by the second operand and the 16-bit immediate data specified by the third operand. Stores the result in the 16-bit register specified by the first operand.

Bytes: 4/5/6
Clocks:
46 to 52, µPD70108
46 to 52, µPD70116 odd addresses
42 to 48, µPD70116 even addresses
Transfers: 1
Flag operation:

\[
\begin{array}{cccccccc}
V & S & Z & AC & P & CY \\
X & U & U & U & U & X \\
\end{array}
\]

Example:
MUL CW,WORD_VAR,200H
\; ;CW = [WORD_VAR]*200H
MUL AW,[IX],850
\; ;AW = [IX]*850
DIVISION

DIVU reg8
Divide unsigned, 8-bit register

| temp ← AW |
| When temp ÷ reg ≥ FFH: |
| AH ← temp % reg8 |
| AL ← temp ÷ reg8 |

| When temp ÷ reg > FFH: |
| (SP−1,SP−2) ← PSW, |
| (SP−3,SP−4) ← PS, |
| (SP−5,SP−6) ← PC, |
| SP ← SP − 6, |
| IE ← 0, |
| BRK ← 0, |
| PS ← (003H, 002H), |
| PC ← (001H, 000H) |

Divides (using unsigned division) the contents of the AW 16-bit register by the contents of the 8-bit register specified by the operand. The resulting quotient is stored in the AL register. Any remainder is stored in the AH register.

When the quotient exceeds FFH (the capacity of the AL destination register) the vector 0 interrupt is generated. When this occurs, the quotient and remainder become undefined. This usually occurs when the divisor is 0. The fractional quotient is rounded off.

Bytes: 2
Clocks: 19
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

Example:

MOV AW,204
MOV CL,10
DIVU CL

;AL = 20, AH = 4

DIVU mem8
Divide unsigned, 8-bit memory

| temp ← AW |
| When temp ÷ (mem8) = FFH: |
| AH ← temp % (mem8), |
| AL ← temp ÷ (mem8), |

| When temp ÷ (mem8) > FFH: |
| (SP−1,SP−2) ← PSW, |
| (SP−3,SP−4) ← PS, |
| (SP−5,SP−6) ← PC, |
| SP ← SP − 6, |
| IE ← 0, |
| BRK ← 0, |
| PS ← (003H, 002H), |
| PC ← (001H, 000H), |

Divides (using unsigned division) the contents of the AW 16-bit register by the 8-bit memory contents specified by the operand. The quotient is stored in the AL register and the remainder, if any, is stored in the AH register.

When the quotient exceeds FFH — the capacity of the AL destination register — the vector 0 interrupt is generated. When this occurs, the quotient and remainder become undefined. This especially occurs when the divisor is 0. The fractional quotient is rounded off.

Bytes: 2/3/4
Clocks: 25
Transfers: 1
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

Example:

MOV AW,3410
MOV [BW],19
DIVU [BW]

;AL = 179, AH = 9
DIVU reg16
Divide unsigned, 16-bit register

7 1 1 1 1 0 1 1 1
1 1 1 1 0 reg

temp ← DW,AW
When temp ÷ reg16 > FFFFH:
(SP−1,SP−2) ← PSW,
(SP−3,SP−4) ← PS,
(SP−5,SP−6) ← PC,
SP ← SP − 6
IE ← 0,
BRK ← 0,
PS ← (003H, 002H),
PC ← (001H, 000H)
All other times:
 DW ← temp % reg16, AW ← temp ÷ reg16

Divides (using unsigned division) the contents of the DW and AW 16-bit register pair by the contents of the 16-bit register specified by the operand. The quotient is stored in the AW register. The remainder, if any, is stored in the DW register. When the quotient exceeds FFFFH (the capacity of the AW destination register) the vector 0 interrupt is generated, and the quotient and remainder become undefined. This most often occurs when the divisor is 0. The fractional quotient is rounded off.

Bytes: 2
Clocks: 25
Transfers: None
Flag operation:

V S Z AC P CY
U U U U U U

Example:
MOV DW,0348H
MOV AW,2197H
; DW,AW = 03482197H
MOV BW,2000H
DIVU BW
; AW = 1A41H, DW = 0197H

DIVU mem 16
Divide unsigned, 16-bit memory

7 1 1 1 1 0 1 1 1
1 1 1 1 0 mem

mod 1 1 0 mem
(disp-low)
(disp-high)

temp ← DW,AW
When temp ÷ (mem16) > FFFFH:
(SP−1,SP−2) ← PSW,
(SP−3,SP−4) ← PS,
(SP−5,SP−6) ← PC,
SP ← SP − 6
IE ← 0,
BRK ← 0,
PS ← (003H, 002H),
PC ← (001H, 000H)
All other times:
 DW ← temp % (mem16), AL ← temp ÷ (mem16)

Divides (using unsigned division) the contents of the DW and AW 16-bit register pair by the 16-bit memory contents specified by the operand. The quotient is stored in the AW register. The remainder, if any, is stored in the DW register.

When the quotient exceeds FFFFH (the capacity of the AW destination register) the vector 0 interrupt is generated and the quotient and remainder become undefined. This especially occurs when the divisor is 0. The fractional quotient is rounded off.

Bytes: 2/3/4
Clocks: 35, µPD70108 35, µPD70116 odd addresses 31, µPD70116 even addresses
Transfers: 1
Flag operation:

V S Z AC P CY
U U U U U U

Example:
MOV DW,0
MOV AW,100
MOV [IX][BX],5
DIVU [IX][BX]
; AW = 0014H = 20, DW = 0
DIV reg8
Divide signed, 8-bit register

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
7 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

**temp ← AW**

When temp ÷ reg8 > 0 and temp ÷ reg8 > 7FH or temp ÷ reg8 > 0 and temp ÷ reg8 > 0-7FH-1:

(SP−1,SP−2) ← PSW,
(SP−3,SP−4) ← PS,
(SP−5,SP−6) ← PC,
SP ← SP − 6,
IE ← 0,
BRK ← 0,
PS ← (003H, 002H),
PC ← (001H, 000H)

All other times:

AH ← temp % reg8,
AL ← temp ÷ reg8

Divides (using signed division) the contents of the AW 16-bit register by the contents of the 8-bit register specified by the operand. The quotient is stored in the AL 8-bit register. The remainder, if any, is stored in the AH register. The maximum value of a positive quotient is +127 (7FH), and the minimum value of a negative quotient is −127 (81H).

When a quotient is greater than either maximum value(s) the quotient and remainder become undefined, and the vector 0 interrupt is generated. This especially occurs when the divisor is 0. A fractional quotient is rounded off. The remainder will have the same sign as the dividend.

Bytes: 2
Clocks: 29 to 34 (according to data)
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

Example:

MOV AW, −247
MOV CL, 3
DIV CL

;AL = −82, AH = −1

DIV mem8
Divide signed, 8-bit memory

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
7 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

**temp ← AW**

When temp ÷ (mem8) > 0 and (mem8) > 7FH or temp ÷ (mem8) < 0 and temp ÷ (mem8) > 0-7FH-1:

(SP−1,SP−2) ← PSW,
(SP−3,SP−4) ← PS,
(SP−5,SP−6) ← PC,
SP ← SP − 6,
IE ← 0,
BRK ← 0,
PS ← (003H, 002H),
PC ← (001H, 000H),

All other times:

AH ← temp % (mem8), AL ← temp ÷ (mem8)

Divides (using signed division) the contents of the AW 16-bit register by the contents of the 8-bit memory location specified by the operand. The quotient is stored in the 8-bit AL register, while the remainder, if any, is stored in the AH register. The maximum value of a positive quotient is +127 (7FH), and the minimum value of a negative quotient is −127 (81H). When a quotient is greater than either maximum value(s), the quotient and remainder become undefined and the vector 0 interrupt is generated.

This especially occurs when the divisor is 0. A fractional quotient is rounded off. The remainder will have the same sign as the dividend.

Bytes: 2/3/4
Clocks: 35 to 40 (according to data)
Transfers: 1
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

Example:

MOV AW, 1234
MOV [BW], −20
DIV [BW]

;AL = −61, AH = 14
DIV reg16
Divide signed, 16-bit register

| 7 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

temp ← DW,AW
When temp ÷ reg16 > 0 and temp ÷ reg16 < 7FFFH or temp ÷ reg16 < 0 and temp ÷ reg16 > 0-7FFFH-1:
(SP−1,SP−2) ← PSW,
(SP−3,SP−4) ← PS,
(SP−5,SP−6) ← PC,
SP ← SP − 6,
IE ← 0,
BRK ← 0,
PS ← (003H, 002H),
PC ← (001H, 000H)
All other times:
DW ← temp % reg16, AW ← temp ÷ reg16

Divides (using signed division) the contents of the DW and AW 16-bit register pair by the contents of the 16-bit register specified by the operand. The quotient is stored in the AW 16-bit register, while the remainder, if any, is stored in the DW register. The maximum value of a positive quotient is +32,767 (7FFFH) and the minimum value of a negative quotient is −32,767 (8001H). When the quotient is greater than either maximum value(s), the quotient and remainder become undefined, and the vector 0 interrupt is generated. This especially occurs when the divisor is 0. A fractional quotient is rounded off. The remainder will have the same sign as the dividend.

Bytes: 2
Clocks: 38 to 43 (according to the data)
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

Example:

```
MOV DW,0123H
MOV AW,4567H
MOV CW,1000H
DIV CW

;AW = 1234H, DW = 0567H
```
DIV mem16
Divide signed, 16-bit memory

```
  7  0
1 1 1 1 0 1 1 1
  mem
```

(temp ← DW,AW)

When temp ÷ (mem16) > 0 and temp ÷ (mem16) < 7FFFH
or temp ÷ (mem16) < 0 and temp ÷ (mem16) > 0-7FFFH-1:

(SP-1,SP-2) ← PSW,
(SP-3,SP-4) ← PS,
(SP-5,SP-6) ← PC,
SP ← SP - 6,
IE ← 0,
BRK ← 0,
PS ← (003H, 002H),
PC ← (001H, 000H)

All other times:

DW ← temp % (mem16), AW ← temp ÷ (mem16)

Divides (using signed division) the contents of the DW and the AW 16-bit register pair by the contents of the 16-bit memory location specified by the operand. The quotient is stored in the AW 16-bit register, while the remainder, if any, is stored in the DW register. The maximum value of a positive quotient is +32,767 (7FFFH), and the minimum value of a negative quotient is -32,767 (8001H). When the quotient is greater than either maximum value(s), the quotient and remainder become undefined and the vector 0 interrupt is generated. This especially occurs when the divisor is 0. A fractional quotient is rounded off. The remainder will have the same sign as the dividend.

Bytes: 2/3/4

Clocks:
48 to 53, μPD70108
48 to 53, μPD70116 odd addresses
44 to 49, μPD70116 even addresses

Transfers: 1

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

Example:

```
MOV DW,0
MOV AW, -34
MOV [IY], -2
DIV [IY]
;AW = 17, DW = 0
```
BCD ADJUST

ADJBA (no operand)

Adjust byte add

```
    7     0
   0 0 1 1 0 1 1 1
```

Adjusts the result of unpacked decimal addition stored in the AL register into a single unpacked decimal number. The higher 4 bits become zero.

When AL AND 0FH > 9 or AC=1:
- AL ← AL + 6,
- AH ← AH + 1,
- AC ← 1,
- CY ← AC,
- AL ← AL AND 0FH

Bytes: 1
Clocks: 3
Transfers: None
Flag operation:

```
 V S Z AC P CY
U U U X U X
```

Example: ADJBA

---

ADJ4A (no operand)

Adjust Nibble Add

```
7 0
0 0 1 0 0 1 1 1
```

When AL AND 0FH < 9 or AC=1:
- AL ← AL + 6,
- CY ← CY OR AC,
- AC ← 1

When AL > 9FH or CY=1:
- AL ← AL + 60H,
- CY ← 1

Adjusts the result of packed decimal addition stored in the AL register into a single packed decimal number.

Bytes: 1
Clocks: 3
Transfers: None
Flag operation:

```
 V S Z AC P CY
X X X X X X
```

Example: ADJ4A
### ADJBS (no operand)

Adjust byte subtract

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

When AL AND 0FH > 9 or AC=1:
- AL ← AL - 6,
- AH ← AH - 1,
- AC ← AC,
- CY ← AC,
- AL ← AL AND 0FH

Adjust the result of unpacked decimal subtraction stored in the AL register into a single unpacked decimal number. The higher 4 bits become zero.

- Bytes: 1
- Clocks: 7
- Transfers: None
- Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>X</td>
<td>U</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: ADJBS

### ADJ4S (no operand)

Adjust nibble subtract

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

When AL AND 0FH > 9 or AC=1:
- AL ← AL - 6,
- CY ← AC OR CY,
- AC ← AC,
- AL ← AL AND 0FH

When AL > 9FH or CY=1:
- AL ← AL - 60H,
- CY ← 1

Adjusts the result of packed decimal subtraction stored in the AL register into a single packed decimal number.

- Bytes: 1
- Clocks: 7
- Transfers: None
- Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: ADJ4S
DATA CONVERSION

CVTBD (no operand)

Convert binary to decimal

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

AH ← AL ÷ 0AH
AL ← AL ...% 0AH

Converts the binary 8-bit value in the AL register into a two-digit unpacked decimal number.

The quotient of AL divided by 10 is stored in the AH register. The remainder of this operation is stored in the AL register.

Bytes: 2
Clocks: 15
Transfers: None
Flag operation:

Example: CVTBD

CVTDB (no operand)

Convert decimal to binary

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

AL ← AH × 0AH + AL
AH ← 0

Converts a two-digit unpacked decimal number in the AH and AL registers into a single 16-bit binary number. The value in the AH is multiplied by 10. The product is added to the contents of the AL register and the result is stored in AL. AH becomes 0.

Bytes: 2
Clocks: 7
Transfers: None
Flag operation:

Example: CVTDB
CVTBW (no operand)
Convert byte to word

|    | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
When AL < 80H:
   AH ← 0
All other times:
   AH ← FFH
Expands the sign of the byte in the AL register to the AH register. Use this instruction to produce a double-length (word) dividend from a byte before a byte division is performed.
Bytes: 1
Clocks: 2
Transfers: None
Flag operation: None
Example: CVTBW

CVTWL (no operand)
Convert word to long word

|    | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
When AW < 8000H:
   DW ← 0
All other times:
   DW ← FFFFH
Expands the sign of the word in the AW register to the DW register. Use this instruction to produce a double-length (double-word) dividend from a word before a word division is performed.
Bytes: 1
Clocks: 4 or 5 (according to data)
Transfers: None
Flag operation: None
Example: CVTWL
**COMPARISON**

**CMP reg,reg**

Compare register and register

```
   7  0
0 0 1 1 1 0 1 1 W
1 1  reg  reg
```

reg − reg

Subtracts the contents of the 8- or 16-bit register specified by the second operand from the contents of the 8- or 16-bit register specified by the first operand. The result is not stored and only the flags are affected.

Bytes: 2
Clocks: 2
Transfers: None

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

CMP AW,BW

CMP CH,DL

---

**CMP mem,reg**

Compare memory and register

```
   7  0
0 0 1 1 1 0 0 W
0 0 1 1 0 0
1 1 1 1 reg mem
```

(mem) − reg

Subtracts the contents of the 8- or 16-bit register specified by the second operand from the 8- or 16-bit memory contents addressed by the first operand. The result is not stored and only the flags are affected.

Bytes: 2/3/4
Clocks:
- When W=0: 11
- When W=1: 15, μPD70108 15, μPD70116 odd addresses
  11, μPD70116 even addresses

Transfers: 1

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

CMP WORD_VAR,IX

CMP BYTE_VAR,CL

CMP [BW],AH
\textbf{CMP reg,mem}

Compare register and memory

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
7 & 6 & 5 & 4 & 3 & 0 \hline
0 & 0 & 1 & 1 & 1 & 0 & 1 & W \hline
\end{tabular}
\end{center}

\begin{tabular}{|c|c|}
\hline
mod & reg \hline
\end{tabular}

\begin{tabular}{|c|}
\hline
(disp-low) \hline
\end{tabular}

\begin{tabular}{|c|}
\hline
(disp-high) \hline
\end{tabular}

Subtracts the 8- or 16-bit memory contents addressed by the second operand from the contents of the 8- or 16-bit register specified by the first operand. The result is not stored and only the flags are affected.

\textit{reg – (mem)}

Bytes: 2/3/4

Clocks:
- When \( W=0 \): 11
- When \( W=1 \): 15, \( \mu \text{PD70108} \)
  - 15, \( \mu \text{PD70116} \) odd addresses
  - 11, \( \mu \text{PD70116} \) even addresses

Transfers: 1

Flag operation:

\begin{tabular}{|c|c|c|c|c|c|}
\hline
V & S & Z & AC & P & CY \hline
X & X & X & X & X & X \hline
\end{tabular}

Example:
- CMP \( \text{AW,[IX]} \)
- CMP \( \text{CH,BYTE\_VAR} \)

\textbf{CMP reg,imm}

Compare register and immediate data

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
7 & 6 & 5 & 4 & 3 & 0 \hline
0 & 1 & 0 & 0 & 0 & 0 & 0 & S & W \hline
\end{tabular}
\end{center}

\begin{tabular}{|c|}
\hline
1 & 1 & 1 & 1 & 1 & \text{reg} \hline
\end{tabular}

\begin{tabular}{|c|}
\hline
\text{imm8 or imm16-low} \hline
\end{tabular}

\begin{tabular}{|c|}
\hline
\text{imm16-high} \hline
\end{tabular}

\textit{reg – imm}

Subtracts the 8- or 16-bit immediate data specified by the second operand from the contents of the 8- or 16-bit register specified by the first operand. The result is not stored and only the flags are affected.

Bytes: 3/4

Clocks: 4

Transfers: None

Flag operation:

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
V & S & Z & AC & P & CY \hline
X & X & X & X & X & X \hline
\end{tabular}

Example:
- CMP \( \text{BL,5} \)
- CMP \( \text{DW,1200H} \)
### CMP mem, imm

Compare memory and immediate data

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **mod**: 1 1 1 mem
- **(disp-low)**
- **(disp-high)**
- **imm8 or imm16-low**
- **imm16-high**

(mem) — imm

Subtracts the 8- or 16-bit immediate data specified by the second operand from the 8- or 16-bit memory contents addressed by the first operand. The result is not stored and only the flags are affected.

- **Bytes**: 3/4/5/6
- **Clocks**: When $W=0$: 13, When $W=1$: 17, $\mu$PD70108
- **Transfers**: 1
- **Flag operation**:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Example**:  
CMP BYTE PTR [BW], 3  
CMP WORD_VAR, 7000H

### CMP acc, imm

Compare accumulator and immediate data

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>W</td>
<td></td>
</tr>
</tbody>
</table>

- **imm8 or imm16-low**
- **imm16-high**

When $W=0$: AL — imm8  
When $W=1$: AW — imm16

Subtracts the 8- or 16-bit immediate data specified by the second operand from the accumulator (AL or AW) specified by the first operand. The result is not stored and only the flags are affected.

- **Bytes**: 2/3
- **Clocks**: 4
- **Transfers**: None
- **Flag operation**:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Example**:  
CMP AL, 0  
CMP AW, 800H
COMPLEMENT OPERATION

NOT reg

Not register

| reg —→ reg |
|---|---|
| 7 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | W |
| 1 | 1 | 0 | 1 | 0 | reg |

Inverts (by performing a 1’s complement) each bit of the 8- or 16-bit register specified by the operand and stores the result in the specified register.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation: None

Example:

NOT BW
NOT CL

NOT mem

Not memory

<table>
<thead>
<tr>
<th>mem</th>
<th>—→ (mem)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mod</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(disp-low)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(disp-high)</td>
</tr>
</tbody>
</table>

Inverts (by performing a 1’s complement) each bit of the 8- or 16-bit memory location addressed by the operand and stores the result in the addressed memory location.

Bytes: 2/3/4
Clocks:
When W=0: 16
When W=1: 24, μPD70108
24, μPD70116 odd addresses
16, μPD70116 even addresses
Transfers: 2
Flag operation: None

Example:

NOT WORD_VAR[IX][2]
NOT BYTE PTR [IY]
NEG reg
Negate register

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

reg ← reg + 1
Takes the 2's complement of the contents of the 8- or 16-bit register specified by the operand.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY*</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Note: * = 0 if the contents of the operand register is 0.

Example:
NEG BL
NEG AW

NEG mem
Negate memory

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(mod | 0 | 1 | 1 | mem |

| (disp-low) |

| (disp-high) 

(mem) ← (mem) + 1
Takes the 2's complement of the 8- or 16-bit memory contents addressed by the operand.

Bytes: 2/3/4
Clocks:
When W=0: 16
When W=1: 24, µPD70108
24, µPD70116 odd addresses
16, µPD70116 even addresses
Transfers: 2
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY*</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Note: * = 0 if the contents of the memory operand is 0.

Example:
NEG WORD_VAR
NEG BYTE PTR [BW][IX]
LOGICAL OPERATION

TEST reg, reg
Test register and register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

reg AND reg
ANDs the contents of the 8- or 16-bit register specified by the first operand and the 8- or 16-bit register specified by the second operand. The result is not stored and only the flags are affected.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Example:
TEST AW,CW
TEST CL,AH

TEST mem, reg or TEST reg, mem
Test register and memory

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>mod</td>
<td>reg</td>
</tr>
</tbody>
</table>

(mem) AND reg
ANDs the contents of the 8- or 16-bit second operand and the contents of the 8- or 16-bit first operand.
The result is not stored and only the flags are affected.

Bytes: 2/3/4
Clocks:
When W=0: 10
When W=1: 14, µPD70108
14, µPD70116 odd addresses
10, µPD70116 even addresses
Transfers: 1
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Example:
TEST BYTE_VAR, DL
TEST AH, [IX]
**TEST reg,imm**

Test immediate data and register

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>reg</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

imm8 or imm16-low

imm16-high

reg AND imm

ANDs the contents of the 8- or 16-bit register specified by the first operand and the 8- or 16-bit immediate data specified by the second operand. The result is not stored and only the flags are affected.

Bytes: 3/4

Clocks: 4

Transfers: None

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Example:

TEST CW,1
TEST AL,50H

**TEST mem,imm**

Test immediate data and memory

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(disp-low)

(disp-high)

imm8 or imm16-low

imm16-high

(mem) AND imm

ANDs the 8- or 16-bit memory contents addressed by the first operand and the 8- or 16-bit immediate data specified by the second operand. The result is not stored and only the flags are affected.

Bytes: 3/4/5/6

Clocks:

When W=0: 11
When W=1: 15, µPD70108
15, µPD70116 odd addresses
11, µPD70116 even addresses

Transfers: 1

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Example:

TEST BYTE PTR [BW],80H
TEST WORD_VAR,00FFH
TEST acc,imm

Test immediate data and accumulator

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

imm8 or imm16-low

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>W</td>
</tr>
</tbody>
</table>

imm16-high

When W=0: AL AND imm8
When W=1: AW AND imm16

ANDs the contents of the accumulator (AL or AW) specified by the first operand and the 8- or 16-bit immediate data specified by the second operand. The result is not stored and only the flags are affected.

Bytes: 2/3
Clocks: 4
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Example:

TEST AL,12H
TEST AW,8000H

AND reg,reg

AND register with register to register

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

reg ← reg AND reg

ANDs the contents of the 8- or 16-bit register specified by the first operand and the contents of the 8- or 16-bit register specified by the second operand. Stores the result in the register specified by the first operand.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: AND IX,AW
AND mem,reg

AND memory with register to memory

\[
\begin{array}{cccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & W \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{mod} & \text{reg} & \text{mem} \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{(disp-low)} \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{(disp-high)} \\
\end{array}
\]

\((\text{mem}) \leftarrow (\text{mem}) \text{ AND reg}\)

ANDs the 8- or 16-bit memory contents addressed by the first operand and the contents of the 8- or 16-bit register specified by the second operand. Stores the result in the memory location addressed by the first operand.

Bytes: 2/3/4

Clocks:
- When \(W=0\): 16
- When \(W=1\): 24, \(\mu\)PD70108
- 24, \(\mu\)PD70116 odd addresses
- 16, \(\mu\)PD70116 even addresses

Transfers: 2

Flag operation:

\[
\begin{array}{ccccccc}
V & S & Z & AC & P & CY \\
0 & X & X & U & X & 0 \\
\end{array}
\]

Example:

AND [BW][IX]3,AL
AND WORD_VAR,CW

AND reg,mem

AND register with memory to register

\[
\begin{array}{cccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & W \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{mod} & \text{reg} & \text{mem} \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{(disp-low)} \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{(disp-high)} \\
\end{array}
\]

\(\text{reg} \leftarrow \text{reg AND (mem)}\)

ANDs the contents of the 8- or 16-bit register specified by the first operand and the 8- or 16-bit memory contents addressed by the second operand. Stores the result in the register specified by the first operand.

Bytes: 2/3/4

Clocks:
- When \(W=0\): 11
- When \(W=1\): 15, \(\mu\)PD70108
- 15, \(\mu\)PD70116 odd addresses
- 11, \(\mu\)PD70116 even addresses

Transfers: 1

Flag operation:

\[
\begin{array}{ccccccc}
V & S & Z & AC & P & CY \\
0 & X & X & U & X & 0 \\
\end{array}
\]

Example:

AND CL, BYTE_VAR
AND DW[LY]
### AND reg,imm

**AND register with immediate data to register**

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**immediate data**

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

reg ← reg AND imm

ANDs the contents of the 8- or 16-bit register specified by the first operand and the 8- or 16-bit immediate data specified by the second operand. Stores the result in the register specified by the first operand.

- **Bytes:** 3/4
- **Clocks:** 4
- **Transfers:** None
- **Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

**Example:**

```assembly
AND CL,0FEH
AND DW,14H
```

### AND mem,imm

**AND memory with immediate data to memory**

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>mod</td>
<td>1</td>
</tr>
</tbody>
</table>

**immediate data**

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(mem) ← (mem) AND imm

ANDs the 8- or 16-bit memory contents addressed by the first operand and the 8- or 16-bit immediate data specified by the second operand. Stores the result in the memory location addressed by the first operand.

- **Bytes:** 3/4/5/6
- **Clocks:**
  - When W=0: 18
  - When W=1: 26, μPD70108
  - 26, μPD70116 odd addresses
  - 18, μPD70116 even addresses
- **Transfers:** 2
- **Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

**Example:**

```assembly
AND BYTE PTR [IY],30H
AND [IY],3000H
```
**AND acc,imm**

AND accumulator with immediate data to accumulator

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>W</td>
</tr>
</tbody>
</table>

- imm8 or imm16-low
- imm16-high

When W=0: AL ← AL AND imm8
When W=1: AW ← AW AND imm16

ANDs the contents of the accumulator (AL or AW) specified by the first operand and the 8- or 16-bit immediate data specified by the second operand. Stores the result in the accumulator specified by the first operand.

Bytes: 2/3
Clocks: 4
Transfers: None

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Example:
- AND AL,80H
- AND AW,0FH

**OR reg,reg**

OR register and register to register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>W</td>
</tr>
</tbody>
</table>

- imm16-high

reg ← reg OR reg

ORs the contents of the 8- or 16-bit register specified by the first operand and the contents of the 8- or 16-bit register specified by the second operand. Stores the result in the register specified by the first operand.

Bytes: 2
Clocks: 2
Transfers: None

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Example:
- OR AL,AH
- OR BW,CW
OR mem, reg

OR memory and register to memory

\[
\begin{array}{cccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & W \\
\end{array}
\]

mod | reg | mem

\[
\begin{array}{cccc}
\text{(disp-low)} \\
\text{(disp-high)} \\
\end{array}
\]

(mem) ← (mem) OR reg

ORs the 8- or 16-bit memory contents addressed by the first operand and the contents of the 8- or 16-bit register specified by the second operand. Stores the result in the memory location addressed by the first operand.

Bytes: 2/3/4

Clocks:
- When W=0: 16
- When W=1: 24, \( \mu \)PD70108
- 24, \( \mu \)PD70116 odd addresses
- 16, \( \mu \)PD70116 even addresses

Transfers: 2

Flag operation:

\[
\begin{array}{cccccccc}
V & S & Z & AC & P & CY \\
0 & X & X & U & X & 0 \\
\end{array}
\]

Example:

OR BYTE_VAR, CL
OR WORD_VAR [BP], AW

OR reg, mem

OR register and memory to register

\[
\begin{array}{cccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & W \\
\end{array}
\]

mod | reg | mem

\[
\begin{array}{cccc}
\text{(disp-low)} \\
\text{(disp-high)} \\
\end{array}
\]

reg ← reg OR (mem)

ORs the contents of the 8- or 16-bit register specified by the first operand and the 8- or 16-bit memory contents addressed by the second operand. Stores the result in the register specified by the first operand.

Bytes: 2/3/4

Clocks:
- When W=0: 11
- When W=1: 15, \( \mu \)PD70108
- 15, \( \mu \)PD70116 odd addresses
- 11, \( \mu \)PD70116 even addresses

Transfers: 1

Flag operation:

\[
\begin{array}{cccccccc}
V & S & Z & AC & P & CY \\
0 & X & X & U & X & 0 \\
\end{array}
\]

Example

OR CL,[IX]
OR CW,WORD_VAR
**NEC**

**Section 12**

**Instruction Set**

---

**OR reg,imm**

OR register with immediate data to register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>W</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>reg</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>imm8 or imm16-low</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>imm16-high</th>
</tr>
</thead>
</table>
```

\[
\text{reg} \leftarrow \text{reg OR imm}
\]

ORs the contents of the 8- or 16-bit register specified by the first operand and the 8- or 16-bit immediate data specified by the second operand. Stores the result in the register specified by the first operand.

- **Bytes:** 3/4
- **Clocks:** 4
- **Transfers:** None
- **Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

**Example:**

```
 OR  CL,80H
 OR  AW,OFH
```

---

**OR mem,imm**

OR memory with immediate data to memory

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>W</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>mem</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>(disp-low)</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>(disp-high)</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>imm8 or imm16-low</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>imm16-high</th>
</tr>
</thead>
</table>
```

\[
\text{(mem) \leftarrow (mem) OR imm}
\]

ORs the 8- or 16-bit memory contents addressed by the first operand and the 8- or 16-bit immediate data specified by the second operand. Stores the result in the memory location addressed by the first operand.

- **Bytes:** 3/4/5/6
- **Clocks:**
  - When W=0: 18
  - When W=1:
    - 26, µPD70108
    - 26, µPD70116 odd addresses
    - 18, µPD70116 even addresses
- **Transfers:** 2
- **Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

**Example:**

```
 OR  BYTE_VAR,2
 OR  WORD PTR [IX],0FH
```
OR acc,imm
OR accumulator with immediate data to accumulator

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

imm8 or imm16-low

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

imm16-high

When W=0: AL ← AL OR imm8
When W=1: AW ← AW OR imm16

ORs the contents of the accumulator (AL or AW) specified by the first operand and the 8- or 16-bit immediate data specified by the second operand. Stores the result in the accumulator specified by the first operand.

Bytes: 2/3
Clocks: 4
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Example:
- OR AL,34H
- OR AW,1

XOR reg,reg
Exclusive OR, register and register to register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

reg ← reg XOR reg

XORs the contents of the 8- or 16-bit register specified by the first operand and the 8- or 16-bit register specified by the second operand. Stores the result in the register specified by the first operand.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Example:
- XOR AL,AH
- XOR CW,BW
XOR mem, reg

Exclusive OR, memory and register to memory

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>W</td>
</tr>
</tbody>
</table>

mod reg mem

(disp-low)

(disp-high)

(mem) ← (mem) XOR reg

XORs the 8- or 16-bit memory contents addressed by the first operand and the contents of the 8- or 16-bit register specified by the second operand. Stores the result in the memory location addressed by the first operand.

Bytes: 2/3/4

Clocks:
- When W=0: 16
- When W=1: 24, μPD70108 24, μPD70116 odd addresses 16, μPD70116 even addresses

Transfers: 2

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Example

XOR [BW], CL
XOR WORD_VAR, BP

XOR reg, mem

Exclusive OR, register and memory to register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>W</td>
</tr>
</tbody>
</table>

mod reg mem

(disp-low)

(disp-high)

reg ← reg XOR (mem)

XORs the contents of the 8- or 16-bit register specified by the first operand and the 8- or 16-bit memory contents addressed by the second operand. Stores the result in the register specified by the first operand.

Bytes: 2/3/4

Clocks:
- When W=0: 11
- When W=1: 15, μPD70108 15, μPD70116 odd addresses 11, μPD70116 even addresses

Transfers: 1

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Example

XOR BH,[IX]
XOR AW,WORD_VAR
XOR reg,imm

Exclusive OR, register with immediate data to register

\[
\begin{array}{cccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & W \\
1 & 1 & 1 & 1 & 0 & & & \text{reg} \\
& & & \text{imm8 or imm16-low} \\
& & & \text{imm16-high} \\
\end{array}
\]

\( \text{reg} \leftarrow \text{reg XOR imm} \)

XORs the contents of the 8- or 16-bit register specified by the first operand and the 8- or 16-bit immediate data specified by the second operand. Stores the result in the register specified by the first operand.

Bytes: 3/4
Clocks: 4
Transfers: None
Flag operation:

\[
\begin{array}{cccccccc}
V & S & Z & AC & P & CY \\
0 & X & X & U & X & 0 \\
\end{array}
\]

Example

XOR CL,2
XOR IX,OFF00H

XOR mem,imm

Exclusive OR, memory with immediate data to memory

\[
\begin{array}{cccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & W \\
mod & 1 & 1 & 0 & & & & \text{mem} \\
& & & \text{imm8 or imm16-low} \\
& & & \text{imm16-high} \\
& & & (\text{disp-low}) \\
& & & (\text{disp-high}) \\
\end{array}
\]

\( \text{(mem)} \leftarrow (\text{mem}) \text{ XOR imm} \)

XORs the 8- or 16-bit memory contents addressed by the first operand and the 8- or 16-bit immediate data specified by the second operand. Stores the result in the memory location addressed by the first operand.

Bytes: 3/4/5/6
Clocks:
  - When \( W=0 \): 18
  - When \( W=1 \): 26, \( \mu PD70108 \)
  - 26, \( \mu PD70116 \) odd addresses
  - 18, \( \mu PD70116 \) even addresses
Transfers: 2
Flag operation:

\[
\begin{array}{cccccccc}
V & S & Z & AC & P & CY \\
0 & X & X & U & X & 0 \\
\end{array}
\]

Example:

XOR BYTE PTR [Y],OFH
XOR WORD_VAR,OFH
XOR acc,imm

Exclusive OR, accumulator with immediate data to accumulator

XORs the contents of the accumulator (AL or AW) specified by the first operand and the 8- or 16-bit immediate data specified by the second operand. Stores the result in the accumulator specified by the first operand.

When W=0: AL ← AL XOR imm8
When W=1: AW ← AW XOR imm16

Bytes: 2/3
Clocks: 4
Transfers: None
Flag operation:

Example:
XOR AL,0FFH
XOR AW,8000H

BIT MANIPULATION

TEST1 reg8,CL

Test bit CL of the 8-bit register

When bit CL of reg8=0: Z ← 1
When bit CL of reg8=1: Z ← 0
Sets the Z flag to 1 when bit CL of the 8-bit register (specified by the first operand) is 0. Resets the Z flag to 0 when bit CL is 1. Only the lower 3 bits of CL are used to address the bit.

Bytes: 3
Clocks: 3
Transfers: 1
Flag operation:

Example: TEST1 AL,CL
**TEST1 mem8,CL**

Test bit CL of the 8-bit memory

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
| mod | 0 | 0 | 0 | mem

(disp-low)

(disp-high)

When bit CL of (mem8) = 0: Z ← 1
When bit CL of (mem8) = 1: Z ← 0

Sets the Z flag to 1 when bit CL of the 8-bit memory (addressed by the first operand) is 0. Resets the Z flag to 0 when the CL bit is 1. Only the lower 3 bits of CL are used to address the bit.

Bytes: 3/4/5
Clocks: 12
Transfers: 1
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U</td>
<td>X</td>
<td>U</td>
<td>U</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: TEST1 BYTE PTR [BW],CL

**TEST1 reg16,CL**

Test bit CL of the 16-bit register

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>reg</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

When bit CL of reg16 = 0: Z ← 1
When bit CL of reg16 = 1: Z ← 0

Sets the Z flag to 1 when bit CL of the 16-bit register (specified by the first operand) is 0. Resets the Z flag to 0 when the bit is 1. Only the lower 4 bits of CL are used to address a bit.

Bytes: 3
Clocks: 3
Transfers: 1
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U</td>
<td>X</td>
<td>U</td>
<td>U</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: TEST1 AW,CL
### TEST1 mem16, CL
Test bit CL of the 16-bit memory

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

mod 0 0 0 mem

(disp-low)

(disp-high)

When bit CL of (mem16) = 0: \( Z \leftarrow 1 \)
When bit CL of (mem16) = 1: \( Z \leftarrow 0 \)

The first operand specifies the 16-bit memory location and the second operand (CL) specifies the bit position. When the bit specified by CL is 0, the \( Z \) flag is set to 1. When that bit is 1, the \( Z \) flag is reset to 0. Only the lower 4 bits of CL are used to address a bit.

Bytes: 3/4/5

Clocks:
- 16, \( \mu P D 70108 \)
- 16, \( \mu P D 70116 \) odd addresses
- 12, \( \mu P D 70116 \) even addresses

Transfers: 1

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U</td>
<td>X</td>
<td>U</td>
<td>U</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: TEST1 WORD PTR [BW], CL

### TEST1 reg8, imm3
Test bit imm3 of the 8-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |

reg 1 1 0 0 0

imm3

When bit imm3 of reg8 = 0: \( Z \leftarrow 1 \)
When bit imm3 of reg8 = 1: \( Z \leftarrow 0 \)

Sets the \( Z \) flag to 1 when bit imm3 of the 8-bit register (specified by the first operand) is 0. Resets the \( Z \) flag to 0 when the bit is 1. Only the lower 3 bits of the immediate data are used to identify a bit.

Bytes: 4

Clocks: 4

Transfers: None

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U</td>
<td>X</td>
<td>U</td>
<td>U</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: TEST1 BH,1
**TEST1 mem8,imm3**

Test bit imm3 of the 8-bit memory

When bit imm3 of (mem8) = 0:  \( Z \leftarrow 1 \)
When bit imm3 of (mem8) = 1:  \( Z \leftarrow 0 \)

The first operand specifies the 8-bit memory location and the second operand (imm3) specifies the bit position. When the bit specified by imm3 is 0, the Z flag is set to 1. When that bit is 1, the Z flag is reset to 0. Only the lower 3 bits of the immediate data are used to address a bit.

**TEST1 reg16, imm4**

Test bit imm4 of the 16-bit register

When bit imm4 of reg16 = 0:  \( Z \leftarrow 1 \)
When bit imm4 of reg16 = 1:  \( Z \leftarrow 0 \)

The first operand specifies the 16-bit register and the second operand (imm4) specifies the bit position. When the bit specified by imm4 is 0, the Z flag is set to 1. When that bit is 1, the Z flag is reset to 0. Only the lower 4 bits of the immediate data are used to address a bit.

**Flag operation:**

```
<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U</td>
<td>X</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>
```

Example: TEST1 AW,15
NEC

Section 12
Instruction Set

TEST1 mem16,imm4
Test bit imm4 of the 16-bit memory

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

When bit imm4 of (mem16) = 0:  Z ← 1
When bit imm4 of (mem16) = 1:  Z ← 0

The first operand specifies the 16-bit memory and the second operand (imm4) specifies the bit position. When the bit specified by imm4 is 0, the Z flag is set to 1. When that bit is 1, the Z flag is reset to 0. The immediate data in the last byte of the instruction is valid only for the lower 4 bits.

Bytes: 4/5/6
Clocks: 17, μPD70108
17, μPD70116 odd addresses
13, μPD70116 even addresses

Transfers: 1
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U</td>
<td>X</td>
<td>U</td>
<td>U</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: TEST1 WORD PTR [BP],8

NOT1 reg8,CL
Not bit CL of the 8-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Bit CL of reg8 ← bit CL of reg8

The CL register (second operand) specifies which bit of the 8-bit register (specified by the first operand) is to be inverted. Only the lower 3 bits of the CL register are used.

Bytes: 3
Clocks: 4
Transfers: None
Flag operation: None
Example: NOT1 BH,CL
NOT1 mem8, CL
Not bit CL of the 8-bit memory

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

mod 0 0 0 mem

(disp-low)

(disp-high)

Bit CL of (mem8) ← bit CL of (mem8)
The CL register (second operand) specifies which bit of the 8-bit memory location (specified by the first operand) is to be inverted. Only the lower 3 bits of the CL register are used.

Bytes: 3/4/5
Clocks: 18
Transfers: 2
Flag operation: None
Example: NOT1 BYTE_VAR, CL

NOT1 reg16, CL
Not bit CL of the 16-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

1 1 0 0 0 reg

Bit CL of reg16 ← bit CL of reg16
The CL register (second operand) specifies which bit of the 16-bit register (specified by the first operand) is to be inverted. Only the lower 4 bits of the CL register are used.

Bytes: 3
Clocks: 4
Transfers: None
Flag operation: None
Example: NOT1 AW, CL
NOT1 mem16,CL
Not bit CL of the 16-bit memory

```
    7 6 5 4 3 2 1 0
0 0 0 0 1 1 1 1
0 0 0 1 0 1 1 1
mod 0 0 0 mem
   (disp-low)
   (disp-high)
```

Bit CL of (mem16) ← bit CL of (mem16)
The CL register (second operand) specifies which bit of the 16-bit memory location (addressed by the first operand) is to be inverted. Only the lower 4 bits of the CL register are used.

Bytes: 3/4/5
Clocks: 26, μPD70108
        26, μPD70116 odd addresses
        18, μPD70116 even addresses

Transfers: 2
Flag operation: None
Example: NOT1  WORD_VAR,CL

---

NOT1 reg8,imm3
Not bit imm3 of the 8-bit register

```
    7 6 5 4 3 2 1 0
0 0 0 0 1 1 1 1
0 0 0 1 1 1 1 0
1 1 0 0 0 reg
   imm3
```

Bit imm3 of reg8 ← bit imm3 of reg8
Bit imm3 (second operand) specifies which bit of the 8-bit register (specified by the first operand) is to be inverted. Only the lower 3 bits of the immediate data at the fourth byte of the instruction are used.

Bytes: 4
Clocks: 5
Transfers: None
Flag operation: None
Example: NOT1  AH,3
NOT1 mem8,imm3
Not bit imm3 of 8-bit memory

Bit imm3 of mem8 ← bit imm3 of mem8

Bit imm3 (second operand) specifies which bit of the 8-bit memory location (addressed by the first operand) is to be inverted. Only the lower 3 bits of the immediate data are used in the last byte of the instruction.

Bytes: 4/5/6
Clocks: 19
Transfers: 2
Flag operation: None
Example: NOT1 BYTE PTR [BW][IX]34H,4

NOT1 reg16,imm4
Not bit imm4 of the 16-bit register

Bit imm4 of reg16 ← bit imm4 of reg16

Bit imm4 (second operand) specifies which bit of the 16-bit register (specified by the first operand) is to be inverted. Only the lower 4 bits of the immediate data are used in the fourth byte of the instruction.

Bytes: 4
Clocks: 5
Transfers: None
Flag operation: None
Example: NOT1 BW,15
Section 12
Instruction Set

NOT1 mem16,imm4
Not bit imm4 of the 16-bit memory

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 1 1 1</td>
<td>0 0 0 1 1 1 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mod</th>
<th>mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>mem</td>
</tr>
</tbody>
</table>

Bit imm4 of (mem16) ← bit imm4 of (mem16)
The bit imm4 (second operand) specifies which bit of the 16-bit memory location (addressed by the first operand) is to be inverted. Only the lower 4 bits of the immediate data are used in the last byte of the instruction.

Bytes: 4/5/6
Clocks:
- 27, μPD70108
- 27, μPD70116 odd addresses
- 19, μPD70116 even addresses

Transfers: 2
Flag operation: None
Example: NOT1 WORD_VAR,0

NOT1 CY
Not carry flag

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

CY ← ¬CY
Inverts the CY flag.
Bytes: 1
Clocks: 2
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>X</td>
</tr>
</tbody>
</table>

Example: NOT1 CY
CLR1 reg8,CL
Clear bit CL of the 8-bit register

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Bit CL of reg8 ← 0
Clears the bit specified by CL of the 8-bit register (specified by the first operand) to 0. Only the lower three bits of CL are used.

Bytes: 3
Clocks: 5
Transfers: None
Flag operation: None
Example: CLR1 AL,CL

CLR1 mem8,CL
Clear bit CL of the 8-bit memory

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Bit CL of (mem8) ← 0
Clears the bit specified by CL of the 8-bit memory location (addressed by the first operand) to 0. Only the lower three bits of CL are used.

Bytes: 3/4/5
Clocks: 14
Transfers: 2
Flag operation: None
Example: CLR1 BYTE_VAR,CL
**CLR1 reg16,CL**

Clear bit CL of the 16-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Bit CL of reg16 ← 0

Clears the bit specified by CL of the 16-bit register (specified by the first operand) to 0. Only the lower four bits of CL are used.

Bytes: 3

Clocks: 5

Transfers: None

Flag operation: None

Example: CLR1 AW,CL

---

**CLR1 mem16,CL**

Clear bit CL of the 16-bit memory

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Bit CL of (mem) ← 0

Clears the bit specified by CL of the 16-bit memory location (addressed by the first operand) to 0. Only the lower 4 bits of CL are used.

Bytes: 3/4/5

Clocks:

22, μPD70108
22, μPD70116 odd addresses
14, μPD70116 even addresses

Transfers: 2

Flag operation: None

Example: CLR1 WORD_VAR,CL
CLR1 reg8,imm3
Clear bit imm3 of the 8-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit imm3 of reg8 ← 0
Clears the bit specified by the 3-bit immediate data (second operand) of the 8-bit register (specified by the first operand) to 0. Only the lower 3 bits of the immediate data are used in the fourth byte of the instruction.

Bytes: 4
Clocks: 6
Transfers: None
Flag operation: None
Example: CLR1 BH,1

CLR1 mem8,imm3
Clear bit imm3 of the 8-bit memory

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit imm3 of (mem8) ← 0
Clears the bit specified by the 3-bit immediate data (second operand) of the 8-bit memory location (addressed by the first operand) to 0. Only the lower 3 bits of immediate data are used in the last byte of the instruction.

Bytes: 4/5/6
Clocks: 15
Transfers: 2
Flag operation: None
Example: CLR1 BYTE_VAR[BW],6
### CLR1 reg16,imm4
Clear bit imm4 of the 16-bit register

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>imm4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>reg</td>
</tr>
</tbody>
</table>

Bit imm4 of reg16 ← 0
Clears the bit specified by the 4-bit immediate data (second operand) of the 16-bit register (specified by the first operand) to 0. Only the lower 4 bits of the immediate data are used in the fourth byte of the instruction.

Bytes: 4
Clocks: 6
Transfers: None
Flag operation: None
Example: CLR1 CW,5

### CLR1 mem16,imm4
Clear bit imm4 of the 16-bit memory

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>imm4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>mem</td>
</tr>
<tr>
<td></td>
<td>mod</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(disp-high)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit imm4 of (mem16) ← 0
Clears the bit specified by the 4-bit immediate data (second operand) of the 16-bit memory location (addressed by the first operand) to 0. Only the lower 4 bits of immediate data are used in the last byte of the instruction.

Bytes: 4/5/6
Clocks: 23, μPD70108
23, μPD70116 odd addresses
15, μPD70116 even addresses
Transfers: 2
Flag operation: None
Example: CLR1 WORD PTR [BP],0
### CLR1 CY

Clear carry flag

<table>
<thead>
<tr>
<th>Bit</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CY ← 0

Cleans the CY flag.

- **Bytes:** 1
- **Clocks:** 2
- **Transfers:** None
- **Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

Example: CLR1 CY

### CLR1 DIR

Clear direction flag

<table>
<thead>
<tr>
<th>Bit</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DIR ← 0

Clears the DIR flag. Sets index registers IX and IY to automatic increment when MOVBK, CMPBK, CMPM, LDM STM, INM, and OUTM are executed.

- **Bytes:** 1
- **Clocks:** 2
- **Transfers:** None
- **Flag operation:**

<table>
<thead>
<tr>
<th>DIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Example: CLR1 DIR Exam
**SET1 reg8,CL**

Set bit CL of the 8-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Bit CL of reg8 $\leftarrow 1$

Sets the bit specified by CL of the 8-bit register (specified by the first operand) to 1. Only the lower three bits of CL are used.

Bytes: 3
Clocks: 4
Transfers: None
Flag operation: None
Example: SET1 BL,CL
Set bit CL of the 8-bit memory

**SET1 mem8,CL**

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mod</td>
<td>0</td>
</tr>
</tbody>
</table>

Bit CL of (mem8) $\leftarrow 1$

Sets the bit specified by CL of the 8-bit memory location (addressed by the first operand) to 1. Only the lower three bits of CL are used.

Bytes: 3/4/5
Clocks: 13
Transfers: 2
Flag operation: None
Example: SET1 BYTE PTR [BW],CL
SET1 reg16, CL
Set bit CL of the 16-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit CL of reg16 $\leftarrow$ 1
Sets the bit specified by CL of the 16-bit register (specified by the first operand) to 1. Only the lower four bits of CL are used.

Bytes: 3
Clocks: 4
Transfers: None
Flag operation: None
Example: SET1 BW, CL

SET1 mem16, CL
Set bit CL of the 16-bit memory

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit CL of (mem16) $\leftarrow$ 1
Sets the bit specified by CL of the 16-bit memory location (addressed by the first operand) to 1. Only the lower 4 bits of CL are used.

Bytes: 3/4/5
Clocks: 21, $\mu$PD70108
21, $\mu$PD70116 odd addresses
13, $\mu$PD70116 even addresses
Transfers: 2
Flag operation: None
Example: SET1 WORD_VAR, CL
SET1 reg8,imm3
Set bit imm3 of the 8-bit register

Bit imm3 of reg8 ← 1
Sets the bit specified by the 8-bit immediate data (second operand) of the 8-bit register (specified by the first operand) to 1. Only the lower 3 bits of the immediate data are used in the fourth byte of the instruction.

Bytes: 4
Clocks: 5
Transfers: None
Flag operation: None
Example: SET1 AL,4

SET1 mem8,imm3
Set bit imm3 of the 8-bit memory

Bit imm3 of (mem8) ← 1
Sets the bit specified by the 3-bit immediate data (second operand) of the 8-bit memory location (addressed by the first operand) to 1. Only the lower 3 bits of the immediate data are used in the last byte of the instruction.

Bytes: 4/5/6
Clocks: 14
Transfers: 2
Flag operation: None
Example: SET1 BYTE_VAR,5
**SET1 reg16,imm4**

Set bit imm4 of the 16-bit register

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>reg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit imm4 of reg16 ← 1
Sets the bit specified by the 4-bit immediate data (second operand) of the 16-bit register (specified by the first operand) to 1. Only the lower 4 bits of the immediate data are used in the 4th byte of the instruction.

- **Bytes:** 4
- **Clocks:** 5
- **Transfers:** None
- **Flag operation:** None
- **Example:** SET1 CW,0

---

**SET1 mem16,imm4**

Set bit imm4 of the 16-bit memory

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>mod</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp-low)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp-high)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit imm4 of (mem16) ← 1
Sets the bit specified by the 4-bit immediate data (second operand) of the 16-bit memory location (addressed by the first operand) to 1. Only the lower 4 bits of immediate data are used in the last byte of the instruction.

- **Bytes:** 4/5/6
- **Clocks:**
  - 22, μPD70108
  - 22, μPD70116 odd addresses
  - 14, μPD70116 even addresses
- **Transfers:** 2
- **Flag operation:** None
- **Example:** SET1 Word_Var,15
SET1 CY
Set carry flag

| 7 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

CY ← 1
Sets the CY flag.

Bytes: 1
Clocks: 2
Transfers: None
Flag operation:

Example: SET1 CY

SET1 DIR
Set direction flag

| 7 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |

Dir ← 1
Sets the DIR flag. Sets index registers IX and IY to auto-decrement when MOVBK, CMPBK, CMPM, LDM STM, INM, and OUTM are executed.

Bytes: 1
Clocks: 2
Transfers: None
Flag operation:

Example: SET1 DIR 1 Exam
**SHIFT**

**SHL reg,1**

Shift left register, single bit

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

CY ← MSB of reg, reg ← reg ×2
When MSB of reg ≠ CY: V ← 1
When MSB of reg = CY: V ← 0

Performs a shift left (1 bit) of the 8- or 16-bit register specified by the first operand. Zero is loaded to the LSB of the specified register and the MSB is shifted to the CY flag. If the sign bit is the same after the shift, the V flag is cleared.

**Example:**

- SHL BH,1
- SHL AW,1

**Bytes:** 2

**Clocks:** 2

**Transfers:** None

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Diagram:**

[Diagram showing the shift operation and flag states]
SHL mem, 1
Shift left memory, single bit

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \text{mod \ 1 \ 0 \ 0 \ 0 \ mem} \]

\( (\text{disp-low}) \)

\( (\text{disp-high}) \)

\[ \text{CY} \leftarrow \text{MSB of (mem)}, \ (\text{mem}) \leftarrow (\text{mem}) \times 2 \]

When \( \text{MSB of (mem)} \neq \text{CY} \): \( \text{V} \leftarrow 1 \)
When \( \text{MSB of (mem)} = \text{CY} \): \( \text{V} \leftarrow 0 \)

Performs a shift left (1 bit) of the 8- or 16-bit memory location addressed by the first operand. Zero is loaded to the addressed memory LSB and the MSB is shifted to the CY flag. If the sign bit (bit 7 or 15) remains the same after the shift, the V flag is cleared.

Bytes: 2/3/4
Clocks:
- When \( W=0 \): 16
- When \( W=1 \): 24, \( \mu \text{PD70108} \)
  - 24, \( \mu \text{PD70116} \) odd addresses
  - 16, \( \mu \text{PD70116} \) even addresses

Transfers: 2
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
- SHL BYTE PTR [IX], 1
- SHL WORD_VAR, 1
**SHL reg, CL**

Shift left register, variable bit

![Binary representation of SHL reg, CL](image)

Temp ← CL, while temp ≠ 0
repeat this operation, CY ← MSB of reg,
reg ← reg × 2, temp ← temp − 1

Performs a shift left of the 8- or 16-bit register specified by the first operand by the number in the CL register. Zero is loaded to the specified register's LSB. MSB is shifted to the CY flag.

- **Bytes:** 2
- **Clocks:** 7 + n, where n = number of shifts
- **Transfers:** None
- **Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**
- SHL CL,CL
- SHL BW,CL
SHL mem, CL
Shift left memory, variable bit

\[
\begin{array}{cccccccc}
\text{7} & \text{6} & \text{5} & \text{4} & \text{3} & \text{2} & \text{1} & \text{0} \\
\text{1} & \text{1} & \text{0} & \text{1} & \text{0} & \text{0} & \text{1} & \text{W} \\
\end{array}
\]

mod \quad 1 \quad 0 \quad 0 \quad \text{mem}

\[
\begin{array}{c}
\text{(disp-low)} \\
\text{(disp-high)}
\end{array}
\]

\[
\text{temp} \leftarrow \text{CL}, \quad \text{while temp} \neq 0,
\]

\[
\text{repeat operation, CY} \leftarrow \text{MSB of (mem)},
\]

\[
\text{(mem)} \leftarrow (\text{mem}) \times 2, \quad \text{temp} \leftarrow \text{temp} - 1
\]

Performs a shift left of the 8- or 16-bit memory location addressed by the first operand by the number in the CL register. Zero is loaded to the addressed memory LSB and the MSB is shifted to the CY flag.

Bytes: 2/3/4
Clocks:
When \( W = 0 \): \( 19 + n \)
When \( W = 1 \): \( 27 + n, \mu\text{PD70108} \)
\( 27 + n, \mu\text{PD70116} \) odd addresses
\( 19 + n, \mu\text{PD70116} \) even addresses
where \( n \) = number of shifts.

Transfers: 2
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

\[
\begin{align*}
\text{SHL} & \quad \text{BYTE PTR [IY],CL} \\
\text{SHL} & \quad \text{WORD PTR [IY],CL}
\end{align*}
\]
SHL reg,imm8
Shift left register, multibit

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>reg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| imm8  |   |   |

Temp ← imm8, while temp ≠ 0,
repeat operation, CY ← MSB of reg,
reg ← reg × 2, temp ← temp − 1

Performs a shift left of the 8- or 16-bit register (specified by the first operand) by the 8-bit immediate data (second operand). Zero is loaded to the specified register's LSB. MSB is shifted to the CY flag.

Bytes: 3
Clocks: 7 + n, where n = number of shifts
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
SHL AH,3
SHL DW,15
SHL mem, imm8

Shift left memory, multibit

```
  7  1  1  0  0  0  0  0  0  W

mod  1  0  0  mem

(disp-low)

(disp-high)

imm8
```

temp ← imm8, while temp ≠ 0,
repeat operation, CY ← MSB of (mem)
(mem) ← (mem) × 2, temp ← temp − 1

Performs a shift left of the 8- or 16-bit memory location
addressed by the first operand by the bits specified by
the 8-bit immediate data (second operand). Zero is
loaded to the specified memory locations’s LSB. The
MSB is shifted to the CY flag.

Bytes: 3/4/5

Clocks:
- When W=0: 19 + n
- When W=1: 27 + n, µPD70108
  27 + n, µPD70116 odd addresses
  19 + n, µPD70116 even addresses
  where n = number of shifts

Transfers: 2

Flag operation:

```
  V  S  Z  AC  P  CY
  U  X  X  U  X  X
```

Example:

- SHL BYTE PTR [IX] [2], 7
- SHL WORD_VAR, 5
**SHR reg,1**
Shift right register, single bit

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>reg</td>
</tr>
</tbody>
</table>

CY ← MSB of reg, reg ← reg ÷ 2
When MSB of reg ≠ bit following MSB of reg: V ← 1
When MSB of reg = bit following MSB of reg: V ← 0

Performs a logical shift right (1 bit) of the 8- or 16-bit register specified by the first operand. Zero is loaded to the MSB of the specified register and the LSB is shifted to the CY flag. If the sign bit (7 or 15) is the same after the shift, the V flag is cleared.

**Bytes:** 2
**Clocks:** 2
**Transfers:** None
**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

- SHR BH,1
- SHR AW,1
SHR mem, 1
Shift right memory, single bit

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

mod | 1 | 0 | 1 | mem |

(disp-low)

(disp-high)

\[ CY \leftarrow \text{MSB of (mem)}, (\text{mem}) \leftarrow (\text{mem}) \div 2 \]
When MSB of (mem) ≠ bit following MSB of (mem):
\[ V \leftarrow 1 \]
When MSB of (mem) = bit following MSB of (mem):
\[ V \leftarrow 0 \]

Performs a logical shift right (1 bit) of the 8- or 16-bit memory location addressed by the first operand. Zero is loaded to the memory location's MSB and the LSB is shifted to the CY flag. If the sign bit (bit 7 or 15) remains the same after the shift, the V flag is cleared.

Bytes: 2/3/4
Clocks:
- When \( W=0 \): 16
- When \( W=1 \): 24, \( \mu \)PD70108
  - 24, \( \mu \)PD70116 odd addresses
  - 16, \( \mu \)PD70116 even addresses

Transfers: 2
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
- SHR BYTE_VAR [BW], 1
- SHR WORD_VAR [IX], 1
**SHR reg,CL**
Shift right register, variable bit

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>reg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`temp ← CL, while temp ≠ 0,`  
repeat operation, `CY ← MSB of reg,`  
`reg ← reg ÷ 2, temp ← temp − 1`

Performs a logical shift right of the 8- or 16-bit register (specified by the first operand) by the number in the CL register. Zero is loaded to the specified register's MSB. The LSB is shifted to the CY flag.

**Bytes:** 2
**Clocks:** 7 + n, where n = number of shifts
**Transfers:** None
**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

- `SHR AL,CL`
- `SHR BW,CL`
**Section 12**

**Instruction Set**

**SHR mem,CL**

Shift right memory, variable bit

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*mod 1 0 1 mem*

*(disp-low)*

*(disp-high)*

temp ← CL, while temp ≠ 0,
repeat operation, CY ← MSB of (mem),
(mem) ← (mem) ÷ 2, temp ← temp − 1

Performs a logical shift right of the 8- or 16-bit memory location (addressed by the first operand) by the number in the CL register. Zero is loaded to the addressed memory MSB and the LSB is shifted to the CY flag.

**Bytes:** 2/3/4

**Clocks:**
- When W=0: 19 + n
- When W=1: 27 + n, μPD70108
  - 27 + n, μPD70116 odd addresses
  - 19 + n, μPD70116 even addresses
  - where n = number of shifts

**Transfers:** 2

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

SHR BYTE_VAR,CL
SHR WORD PTR [IY],CL

---

**Diagram:**

[Diagram of SHR instruction set]
**SHR reg,imm8**

Shift right register, multibit

```
<table>
<thead>
<tr>
<th>Reg</th>
<th>Imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 0 0 W</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 0 1 reg</td>
<td></td>
</tr>
<tr>
<td>imm8</td>
<td></td>
</tr>
</tbody>
</table>
```

- `temp ← imm8, while temp ≠ 0, repeat operation, CY ← MSB of reg, reg ← reg ÷ 2, temp ← temp − 1`

Performs a shift right of the 8- or 16-bit register (specified by the first operand) by the 8-bit immediate data (second operand). Zero is loaded to the specified register’s MSB. The LSB is shifted to the CY flag.

**Example:**

- `SHR BL,6`
- `SHR IX,2`

**Bytes:** 3

**Clocks:**

- `7 + n, where n = number of shifts`

**Transfers:** None

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

12-110
SHR mem, imm8
Shift right memory, multibit

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp-low)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp-high)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

temp ← imm8, while temp ≠ 0,
repeat operation, CY ← MSB of (mem),
(mem) ← (mem) ÷ 2, temp ← temp − 1

Performs a shift right of the 8- or 16-bit memory location (addressed by the first operand) by the bits specified by the 8-bit immediate data (second operand). Zero is loaded to the specified memory location's MSB. The LSB is shifted to the CY flag.

Bytes: 3/4/5

Clocks:
- When W=0: 19 + n
- When W=1: 27 + n, μPD70108
  - 27 + n, μPD70116 odd addresses
  - 19 + n, μPD70116 even addresses

where n = number of shifts

Transfers: 2

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
- SHR BYTE PTR [BW], 2
- SHR WORD_VAR, 13
**SHRA reg,1**

Shift right arithmetic

```
  7  6  5  4  3  2  1  0
  1  1  0  1  0  0  0  0  W
```

MSB of operand does not change

```
  7  6  5  4  3  2  1  0  reg
  1  1  1  1  1  1  1  1
```

CY ← LSB of reg,
reg ← reg ÷ 2, V ← 0

Performs an arithmetic shift right (1 bit) of the 8- or 16-bit register specified by the first operand. A bit with the same value as the original bit is shifted to the specified register’s MSB. The LSB is shifted to the CY flag. The sign remains unchanged after the shift.

**Flag operation:**

```
V S Z AC P CY
0 X X U X X
```

Example:

- SHRA CL,1
- SHRA AW,1

**Bytes:** 2
**Clocks:** 2
**Transfers:** None
SHRA mem,1
Shift right arithmetic, memory, single bit

```
|   | 1 | 1 | 0 | 1 | 0 | 0 | 0 | W |
```

mod 1 1 1 mem

(disp-low)

(disp-high)

CY ← LSB of (mem),
(mem) ← (mem) ÷ 2, V ← 0
MSB of operand does not change

Performs an arithmetic shift right (1 bit) of the 8- or 16-bit memory location addressed by the first operand. A bit with the same value as the original bit is shifted to the memory location's MSB. The LSB is shifted to the CY flag. The sign remains unchanged after the shift.

Bytes: 2/3/4

Clocks
When W=0: 16
When W=1: 24, μPD70108
24, μPD70116 odd addresses
16, μPD70116 even addresses

Transfers: 2

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
SHRA BYTE_VAR,1
SHRA WORD_VAR,1
SHRA reg, CL
Shift right arithmetic, register, variable bit

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

reg

temp ← CL, while temp ≠ 0,
repeat operation, CY ← LSB of reg,
reg ← reg ÷ 2, temp ← temp − 1

Performs an arithmetic shift right of the 8- or 16-bit register (specified by the first operand) by the number of bits specified by the CL register. A bit with the same value as the original bit is shifted to the register’s MSB. The LSB is shifted to the CY flag. The sign remains unchanged after the shift.

Bytes: 2
Clocks: 7 + n, where n = number of shifts
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
SHRA BL, CL
SHRA DW, CL
SHRA mem, CL

Shift right arithmetic, memory, variable bit

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp-low)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp-high)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

temp ← CL, while temp ≠ 0,
repeat operation, CY ← LSB of (mem),
(mem) ← (mem) ÷ 2, temp ← temp - 1,
MSB of operand does not change

Performs an arithmetic shift right of the 8- or 16-bit memory location (addressed by the first operand) by the number of bits specified in the CL register. A bit with the same value as the original bit is shifted to the memory location's MSB. The LSB is shifted to the CY flag. The sign remains unchanged after the shift.

Bytes: 2/3/4

Clocks:
When W=0: 19 + n
When W=1: 27 + n, μPD70108
27 + n, μPD70116 odd addresses
19 + n, μPD70116 even addresses
where n = number of shifts

Transfers: 2

Flag Operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

SHRA BYTE_VAR, CL
SHRA WORD_VAR, CL
SHRA reg,imm8
Shift right arithmetic, register, multibit

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>reg</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm8</td>
</tr>
</tbody>
</table>

temp ← imm8, while temp ≠ 0,
repeat operation, CY ← LSB of reg,
reg ← reg ÷ 2, temp ← temp − 1,
MSB of operand does not change

Performs an arithmetic shift right of the 8- or 16-bit register (specified by the first operand) by the 8-bit immediate data in the second operand. A bit with the same value as the original bit is shifted to the register's MSB. The LSB is shifted to the CY flag. The sign remains unchanged after the shift.

Bytes: 3
Clocks: 7 + n, where n = number of shifts
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
SHRA CL,3
SHRA BW,7
SHRA mem,imm8
Shift right arithmetic, memory, multibit

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp-low)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp-high)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

temp ← imm8, while temp ≠ 0,
repeat this operation, CY ← LSB of (mem),
(mem) ← (mem) ÷ 2, temp ← temp − 1,
MSB of operand does not change

Performs an arithmetic shift right of the 8- or 16-bit memory location (addressed by the first operand) by the number specified by the 8-bit immediate data in the second operand. A bit with the same value as the original bit is shifted to the register’s MSB. The LSB is shifted to the CY flag. The sign remains unchanged after the shift.

Bytes: 3/4/5
Clocks:
When W=0: 19 + n
When W=1: 27 + n, µPD70108
27 + n, µPD70116 odd addresses
19 + n, µPD70116 even addresses
where n = number of shifts

Transfers: 2
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

SHRA BYTE_VAR,5
SHRA WORD_VAR,7
**ROTATE**

**ROL reg,1**

Rotate left, register, single bit

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>reg</td>
</tr>
</tbody>
</table>

CY ← MSB of reg, reg ← reg × 2 + CY
MSB of reg ≠ CY: V ← 1
MSB of reg = CY: V ← 0

Rotates the 8- or 16-bit register specified by the first operand left by one bit. If the MSB changes, the V flag is set. If the MSB stays the same, the V flag is cleared.

**Example:**

- ROL AH,1
- ROL DW,1

**Bytes:** 2

**Clocks:** 2

**Transfers:** None

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
ROL mem,1
Rotate left, memory, single bit

\[
\begin{array}{cccccccc}
7 & & & & & & & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & W \\
\end{array}
\]

mod 0 0 0 0 mem

(disp-low)

(disp-high)

CY \leftarrow \text{MSB of (mem)},
(mem) \leftarrow (mem) \times 2 + CY
MSB of (mem) \neq CY: V \leftarrow 1
MSB of (mem) = CY: V \leftarrow 0

Rotates the 8- or 16-bit memory location (addressed by the first operand) left by one bit. If the MSB changes, the V flag is set; if it stays the same, the V flag is cleared.

Bytes: 2/3/4
Clocks:
- When W=0: 16
- When W=1: 24, µPD70108
  - 24, µPD70116 odd addresses
  - 16, µPD70116 even addresses

Transfers: 2
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

ROL BYTE_VAR,1
ROL WORD PTR [IX][7],1

49-00041A
ROL reg,CL

Rotate left, register, variable bit

<table>
<thead>
<tr>
<th>reg</th>
<th>CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 0 1 W</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0 0 reg</td>
<td></td>
</tr>
</tbody>
</table>

temp ← CL, while temp ≠ 0,
repeat operation, CY ← MSB of reg,
reg ← reg × 2 + CY,
temp ← temp − 1

Rotates the 8- or 16-bit register specified by the first operand left by the number of bits specified by the CL register.

Bytes: 2
Clocks:
7 + n, where n = number of shifts

Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
ROL DL,CL
ROL BP,CL
**ROL mem,CL**

Rotate left, memory, variable bit

<table>
<thead>
<tr>
<th>7</th>
<th>1 1 0 1 0 0 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>(disp-low)</td>
<td></td>
</tr>
<tr>
<td>(disp-high)</td>
<td></td>
</tr>
</tbody>
</table>

Bytes: 2/3/4

Clocks:
- When W=0: $19 + n$
- When W=1: $27 + n$, $\mu PD70108$
  - $27 + n$, $\mu PD70116$ odd addresses
  - $19 + n$, $\mu PD70116$ even addresses

where $n = \text{number of shifts}$

Transfers: 2

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:

- ROL BYTE PTR [IX],CL
- ROL WORD_VAR,CL
ROL reg, imm8
Rotate left, register, multibit

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

temp ← imm8, while temp ≠ 0,
repeat operation, CY ← MSB of reg,
reg ← reg × 2 + CY,
temp ← temp − 1

Rotates the 8- or 16-bit register (specified by the first operand) left by the number of bits specified by the 8-bit immediate data in the second operand. The register's MSB is shifted to the CY flag and to the LSB.

Bytes: 3
Clocks: 7 + n, where n = number of shifts
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
- ROL DH,3
- ROL IY,7
**ROL mem, imm8**

Rotate left, memory, multibit

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
</table>

```
mod   0   0   0   0   mem
```

```
(disp-low)
```

```
(disp-high)
```

```
imm8
```

temp ← imm8, while temp ≠ 0, repeat operation, CY ← MSB of (mem), (mem) ← (mem) × 2 + CY, temp ← temp − 1

Rotates the 8- or 16-bit memory location (addressed by the first operand) left by the number of bits specified by the 8-bit immediate data in the second operand. The memory location’s MSB is shifted to the CY flag and to the LSB.

**Bytes:** 3/4/5

**Clocks:**
- When W=0: 19 + n
- When W=1: 27 + n, µPD70108
  - 27 + n, µPD70116 odd addresses
  - 19 + n, µPD70116 even addresses
  - where n = number of shifts

**Transfers:** 2

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

ROL BYTE_VAR,7

ROL WORD_VAR,2
ROR reg, 1
Rotate right, register, single bit

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>reg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CY ← LSB of reg, reg ← reg ÷ 2,
MSB of reg ← CY
MSB of reg ≠ bit following MSB of reg: V ← 1
MSB of reg = bit following MSB of reg: V ← 0

Rotates the 8- or 16-bit register (specified by the first operand) right by 1 bit. If the MSB of the specified register changes, the overflow flag is set. If the MSB stays the same, the overflow flag is cleared.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example:
ROR AL,1
ROR CW,1
ROR mem,1
Rotate right, memory, single bit

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mod</td>
<td>0</td>
</tr>
<tr>
<td>(disp-low)</td>
<td></td>
</tr>
<tr>
<td>(disp-high)</td>
<td></td>
</tr>
</tbody>
</table>

CY ← LSB of (mem), (mem) ← (mem) ÷ 2
MSB of (mem) ← CY
MSB of (mem) ≠ bit following MSB of (mem): V ← 1
MSB of (mem) = bit following MSB of (mem): V ← 0

Rotates the 8- or 16-bit memory location addressed by the first operand right by 1 bit. If the MSB of the addressed memory changes, the overflow flag is set. If the MSB stays the same, the overflow flag is cleared.

Bytes: 2/3/4
Clocks:
- When W=0: 16
- When W=1: 24, μPD70108
  - 24, μPD70116 odd addresses
  - 16, μPD70116 even addresses

Transfers: 2

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
- ROR BYTE_VAR,1
- ROR WORD PTR [BW],1
ROR reg, CL
Rotate right, register, variable bit

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>reg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

temp ← CL, while CL ≠ 0, repeat operation,
CY ← LSB of reg, reg ← reg ÷ 2,
MSB of reg ← CY,
temp ← temp − 1

Rotates the 8- or 16-bit register (specified by the first operand) right by the number of bits specified by the CL register.

Bytes: 2
Clocks:
7 + n, where n = number of shifts
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
ROR AH, CL
ROR AW, CL
ROR mem, CL
Rotates right, memory, variable bit

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(disp-low)

(disp-high)

temp ← CL, while temp ≠ 0,
repeat operation,
CY ← LSB of (mem), (mem) ← (mem) / 2,
MSB of (mem) ← CY,
Temp ← temp − 1

Rotates the 8- or 16-bit memory location (specified by the first operand) right by the number of bits specified by the CL register.

Bytes: 2/3/4

Clocks:
When W=0: 19 + n
When W=1: 27 + n, µPD70108
27 + n, µPD70116 odd addresses
19 + n, µPD70116 even addresses
where n = number of shifts

Transfers: 2

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
ROR BYTE_VAR, CL
ROR WORD PTR [IX]2, CL
ROR reg,imm8
Rotate right, register, multibit

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>reg</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

imm8

temp ← imm8, while temp ≠ 0,
repeat operation,
CY ← LSB of reg, reg ← reg ÷ 2,
MSB of reg ← CY,
temp ← temp − 1

Rotates the 8- or 16-bit register (specified by the first operand) right by the number of bits specified by the 8-bit immediate data in the second operand. The register’s LSB is shifted to the MSB and the CY flag.

Bytes: 3
Clocks: 7 + n, where n = number of shifts
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
ROR AL,2
ROR IX,3
ROR mem, imm8

Rotate right, memory, multibit

Bytes: 3/4/5

Clocks:
When W=0: 19 + n
When W=1: 27 + n, µPD70108
27 + n, µPD70116 odd addresses
19 + n, µPD70116 even addresses
where n = number of shifts

Transfers: 2

Flag operation:

Example:
ROR BYTE_VAR, 6
ROR WORD_VAR [IX], 7

temp ← imm8, while temp ≠ 0,
repeat operation,
CY ← LSB of (mem), (mem) ← (mem) ÷ 2,
temp ← temp - 1

Rotates the 8- or 16-bit memory location addressed by the first operand right by the number of bits specified by the 8-bit immediate data in the second operand. The memory location's LSB is shifted to the MSB as well as to the CY flag.
**ROLC reg,1**

Rotate left with carry, register, single bit

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>W</td>
</tr>
</tbody>
</table>

tmpcy ← CY, CY ← MSB of reg,
Reg ← reg × 2 + tmpcy,
MSB of reg = CY: V ← 0
MSB of reg ≠ CY: V ← 1

Rotates the 8- or 16-bit register specified by the first operand left, including the CY flag, by one bit. If the register's MSB changes, the V flag is set. If it stays the same, the V flag is cleared.

**Bytes:** 2

**Clocks:** 2

**Transfers:** None

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

ROLC BL,1
ROLC IY,1
ROLC mem,1
Rotate left with carry, memory, single bit

\[
\begin{array}{cccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & W \\
\end{array}
\]

mod 0 1 0 mem

(disp-low)

(disp-high)

\[
\begin{align*}
\text{tmpcy} & \leftarrow \text{CY}, \text{CY} \leftarrow \text{MSB of (mem)}, \\
(\text{mem}) & \leftarrow (\text{mem}) \times 2 + \text{tmpcy}, \\
\text{MSB of (mem)} & = \text{CY}: \ V \leftarrow 0, \\
\text{MSB of (mem)} & \neq \text{CY}: \ V \leftarrow 1
\end{align*}
\]

Rotates the 8- or 16-bit memory location (addressed by the first operand) left by one bit. The rotation includes the CY flag. If the MSB of the memory location changes, the V flag is set. If it stays the same, the V flag is cleared.

Bytes: 2/3/4

Clocks:
- When W=0: 16
- When W=1: 24, \(\mu\text{PD70108}\)
  - 24, \(\mu\text{PD70116}\) odd addresses
  - 16, \(\mu\text{PD70116}\) even addresses

Transfers: 2

Flag operation:

\[
\begin{array}{cccccc}
V & S & Z & AC & P & CY \\
X & - & - & - & - & - \\
\end{array}
\]

Example:

ROLC BYTE_VAR,1
ROLC WORD PTR [IY],1
ROLC reg, CL
Rotate left with carry, register, variable bit

<table>
<thead>
<tr>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>W</td>
</tr>
</tbody>
</table>

7

\[
\text{temp} \leftarrow \text{CL}, \text{while temp} \neq 0,
\text{repeat operation, tempcy} \leftarrow \text{CY},
\text{CY} \leftarrow \text{MSB of reg, reg} \leftarrow \text{reg} \times 2 + \text{tmpcy},
\text{temp} \leftarrow \text{temp} - 1
\]

Rotates the 8- or 16-bit register (specified by the first operand) left by the number in the CL register. Rotation includes the CY flag.

Bytes: 2
Clocks:
\[7 = n, \text{where} \ n = \text{number of shifts}\]
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
ROLC AL, CL
ROLC BW, CL
**ROLC** mem,CL

Rotate left with carry, memory, variable bit

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mod</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(disp-low)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(disp-high)</td>
</tr>
</tbody>
</table>

temp ← CL, while temp ≠ 0,
repeat operation, tmpcy ← CY,
CY ← MSB of (mem),
(mem) ← (mem) × 2 + tmpcy,
temp ← temp - 1

Rotates the 8- or 16-bit memory location (addressed by the first operand) left by the number in the CL register. Rotation includes the CY flag.

**Bytes:** 2/3/4

**Clocks:**
- When W=0: 19 + n
- When W=1: 27 + n, \( \mu P D 70108 \)
  - 27 + n, \( \mu P D 70116 \) odd addresses
  - 19 + n, \( \mu P D 70116 \) even addresses
where \( n \) = number of shifts

**Transfers:** 2

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

- **ROLC** BYTE PTR [IY],CL
- **ROLC** WORD_VAR,CL

**Bytes:** 2/3/4

**Clocks:**
- When W=0: 19 + n
- When W=1: 27 + n, \( \mu P D 70108 \)
  - 27 + n, \( \mu P D 70116 \) odd addresses
  - 19 + n, \( \mu P D 70116 \) even addresses
where \( n \) = number of shifts

**Transfers:** 2

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

- **ROLC** BYTE PTR [IY],CL
- **ROLC** WORD_VAR,CL

**Bytes:** 2/3/4

**Clocks:**
- When W=0: 19 + n
- When W=1: 27 + n, \( \mu P D 70108 \)
  - 27 + n, \( \mu P D 70116 \) odd addresses
  - 19 + n, \( \mu P D 70116 \) even addresses
where \( n \) = number of shifts

**Transfers:** 2

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

- **ROLC** BYTE PTR [IY],CL
- **ROLC** WORD_VAR,CL

**Bytes:** 2/3/4

**Clocks:**
- When W=0: 19 + n
- When W=1: 27 + n, \( \mu P D 70108 \)
  - 27 + n, \( \mu P D 70116 \) odd addresses
  - 19 + n, \( \mu P D 70116 \) even addresses
where \( n \) = number of shifts

**Transfers:** 2

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**

- **ROLC** BYTE PTR [IY],CL
- **ROLC** WORD_VAR,CL

**Bytes:** 2/3/4

**Clocks:**
- When W=0: 19 + n
- When W=1: 27 + n, \( \mu P D 70108 \)
  - 27 + n, \( \mu P D 70116 \) odd addresses
  - 19 + n, \( \mu P D 70116 \) even addresses
where \( n \) = number of shifts

**Transfers:** 2

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
ROLC reg,imm8
Rotate left with carry, register, multibit

Bytes: 3
Clocks: 7 + n, where n = number of shifts
Transfers: None
Flag operation:

Example:
ROLC BL,3
ROLC AW,14

temp ← imm8, while temp ≠ 0,
repeat operation, tmpcy ← CY,
CY ← MSB of reg, reg ← reg × 2 + tmpcy,
temp ← temp − 1

Rotates the 8- or 16-bit register (specified by the first operand) left by the number of bits specified by the 8-bit immediate data of the second operand. Rotation includes the CY flag.
ROLC mem,imm8

Rotate left with carry, memory, multibit

**Bytes:** 3/4/5

**Clocks:**
- When W=0: 19 + n
- When W=1: 27 + n, µPD70108
  - 27 + n, µPD70116 odd addresses
  - 19 + n, µPD70116 even addresses
where n = number of shifts

**Transfers:** 2

**Flag operation:**

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Example:**
- ROLC BYTE_VAR,3
- ROLC WORD_VAR,5

temp ← imm8, while temp ≠ 0,
repeat operation, tmpcy ← CY,
CY ← MSB of (mem),
(mem) ← (mem) × 2 + tmpcy,
temp ← temp − 1

Rotates the 8- or 16-bit memory location (addressed by the first operand) left by the number of bits specified by the 8-bit immediate data of the second operand. Rotation includes the CY flag.
RORC reg, 1
Rotate right with carry, register, single bit

Bytes: 2
Clocks: 2
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
- RORC BH, 1
- RORC BP, 1

tmpcy ← CY, CY ← LSB of reg,
reg ← reg ÷ 2, MSB of reg ← tmpcy,
MSB of reg ≠ bit following MSB of reg: V ← 1,
MSB of reg = bit following MSB of reg: V ← 0

Rotates the 8- or 16-bit register, specified by the first operand, right (including the CY flag) by one bit. If the MSB changes, the V flag is set. If it remains unchanged, the V flag is cleared.
RORC mem,1
Rotate right with carry, memory, single bit

| 7 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | W |
|---|---|---|---|---|---|---|---|
| mod | 0 | 1 | 1 | mem |

(disp-low)

(disp-high)

tmpcy ← CY, CY ← LSB of (mem),
(mem) ← (mem) ÷ 2, MSB of (mem) ← tmpcy,
MSB of (mem) ≠ bit following MSB of (mem): V ← 1
MSB of (mem) = bit following MSB of (mem): V ← 0

Rotates the 8- or 16-bit memory location (addressed by the first operand) right (including the CY flag) by one bit. If the MSB changes, the V flag is set. If it remains unchanged, the V flag is cleared.

Example:
RORC BYTE PTR [BW],1
RORC WORD_VAR [BW] [IX],1

Bytes: 2/3/4
Clocks:
When W=0: 16
When W=1: 24, μPD70108
24, μPD70116 odd addresses
16, μPD70116 even addresses

Transfers: 2
Flag operation:
RORC reg, CL
Rotate right with carry, register, variable bit

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

temp ← CL, while temp ≠ 2,
repeat operation, tmpcy ← CY,
CY ← LSB of reg, reg ← reg ÷ 2
MSB of reg ← tmpcy, temp ← temp − 1,

Rotates the 8- or 16-bit register specified by the first operand right (including the CY flag) by the number in the CL register.

Bytes: 2
Clocks: 7 + n, where n = number of shifts
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
RORC AL, CL
RORC CW, CL
RORC mem, CL
Rotate right with carry, memory, variable bit

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>W</td>
</tr>
</tbody>
</table>

mod 0 1 1 mem

(disp-low)

(disp-high)

temp ← CL, while temp ≠ 0,
repeat operation, tmpcy ← CY,
CY ← LSB of (mem), reg ← reg ÷ 2,
MSB of (mem) ← tmpcy, temp ← temp − 1

Rotates the 8- or 16-bit memory location specified by the first operand right (including the CY flag) by the number in the CL register.

Bytes: 2/3/4

Clocks:
- When W=0: 19 + n
- When W=1: 27 + n, µPD70108
  - 27 + n, µPD70116 odd addresses
  - 19 + n, µPD70116 even addresses
where n = number of shifts

Transfer: 2

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
- RORC BYTE_VAR, CL
- RORC WORD_VAR [BP], CL

Transfers: 2

Flag operation:
RORC reg, imm8
Rotate right with carry, register, multibit

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

temp ← imm8, while temp ≠ 0,
repeat operation, tmpcy ← CY,
CY ← LSB of reg, reg ← reg ÷ 2,
MSB of reg ← tmpcy, temp ← temp − 1

Rotates the 8- or 16-bit register specified by the first operand right (including the CY flag) by the number of bits specified by the 8-bit immediate data of the second operand.

Bytes: 3
Clocks: 7 + n, where n = number of shifts
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Example:
RORC CH,5
RORC BW,10
RORC mem, imm8
Rotate right with carry, memory multibit

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(disp-low)

(disp-high)

imm8

temp ← imm8, while temp ≠ 0,
repeat operation, tmpcy ← CY,
CY ← LSB of (mem), (mem) ← (mem) ÷ 2,
MSB of (mem) ← tmpcy, temp ← temp − 1

Rotates the 8- or 16-bit memory location addressed by the first operand right (including the CY flag) by the number of bits specified by the 8-bit immediate data of the second operand.

Bytes: 3/4/5

Clocks:
- When W=0: 19 + n
- When W=1: 27 + n, µPD70108
  27 + n, µPD70116 odd addresses
  19 + n, µPD70116 even addresses

where n = number of shifts

Transfers: 2

Flag operation:

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
<th>P</th>
<th>CY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example:

1. RORC BYTE_VAR,3
2. RORC WORD PTR [BW],10
SUBROUTINE CONTROL

CALL near-proc
Call, relative, same segment

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

(disp-low)

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

(disp-high)

(SP - 1, SP - 2) ← PC,
SP ← SP - 2,
PC ← PC + disp

Saves the PC to the stack and loads the 16-bit displacement to the PC. Enables calls to any address within the current segment.

Bytes: 3

Clocks:
- 20, µPD70108
- 20, µPD70116 odd addresses
- 16, µPD70116 even addresses

Transfers: 1

Flag operation: None

Example: CALL NEAR_PROC

CALL regptr16
Call, register, same segment

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

(SP - 1, SP - 2) ← PC,
SP ← SP - 2,
PC ← regptr16

Saves the PC to the stack and loads the value of the 16-bit register specified by the operand to the PC. Enables calls to any address within the current segment.

Bytes: 2

Clocks:
- 18, µPD70108
- 18, µPD70116 odd addresses
- 14, µPD70116 even addresses

Transfers: 1

Flag operation: None

Example: CALL BX
CALL memptr16
Call, memory, same segment

\[
\begin{array}{cccccccccc}
7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{mod} & 0 & 1 & 0 \\
\text{mem} & \\
\end{array}
\]

\[
\begin{array}{c}
\text{(disp-low)} \\
\text{(disp-high)} \\
\end{array}
\]

\[(SP - 1, SP - 2) \leftarrow PC,\]
SP \leftarrow SP - 2, PC \leftarrow (memptr16)

Saves the PC to the stack and loads the contents of the
16-bit memory location addressed by the operand to the
PC. Enables calls to any address within the current
segment.
Bytes: 2/3/4
Clocks:
31, \( \mu \)PD70108
31, \( \mu \)PD70116 odd addresses
23, \( \mu \)PD70116 even addresses
Transfers: 2
Flag operation: None
Example: CALL TABLE_ENTRY [IX]

CALL far-proc
Call, direct, external segment

\[
\begin{array}{cccccccccc}
7 & 7 & 7 & 7 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{c}
\text{(offset-low)} \\
\text{(offset-high)} \\
\end{array}
\]

\[
\begin{array}{c}
\text{(seg-low)} \\
\text{(seg-high)} \\
\end{array}
\]

\[(SP - 1, SP - 2) \leftarrow PS,\]
(SP − 3, SP − 4) \leftarrow PC,
SP \leftarrow SP − 4,
PS \leftarrow seg,
PC \leftarrow offset

Saves the PS and PC to the stack. Loads the fourth and
fifth bytes of the instruction to the PS and the second and
third bytes to the PC. Enables calls to any address in any
segment.
Bytes: 5
Clocks:
29, \( \mu \)PD70108
29, \( \mu \)PD70116 odd addresses
21, \( \mu \)PD70116 even addresses
Transfers: 2
Flag operation: None
Example: CALL FAR_PROC
**CALL memptr32**

Call, memory, external segment

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>mem</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp-low)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp-high)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(SP - 1, SP - 2) ← PS,
(SP - 3, SP - 4) ← PC,
SP ← SP - 4,
PS ← (memptr32 + 3, memptr32 + 2),
PC ← (memptr32 + 1, memptr32)

Saves the PS and PC to the stack. Loads the higher two bytes of the 32-bit memory addressed by the operand to the PS. Loads the lower two bytes to the PC. Enables calls to any address in any segment.

Bytes: 2/3/4

Clocks:
- 47, µPD70108
- 47, µPD70116 odd addresses
- 31, µPD70116 even addresses

Transfers: 4

Flag operation: None

Example: CALL FAR_TABLE [IY]

**RET (no operand)**

Return from procedure, same segment

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>
```

PC ← (SP + 1, SP),
SP ← SP + 2

Used for returning from intrasegment calls. Restores the PC from the stack. The assembler automatically distinguishes this instruction from the other RET instruction with no operand.

Bytes: 1

Clocks:
- 19, µPD70108
- 19, µPD70116 odd addresses
- 15, µPD70116 even addresses

Transfers: 1

Flag operation: None

Example: RET
RET pop-value
Return from procedure, SP jump, same segment

```
7 1 1 0 0 0 0 1 0
```

- pop-value-low
- pop-value-high

PC ← (SP + 1, SP),
SP ← SP + 2,
SP ← SP + pop-value

Restores the PC from the stack and adds the 16-bit pop-value specified by the operand. Effective for jumping a desired number of parameters when the parameters saved in the stack become unnecessary to the program. Used for returning from intrasegment calls. The assembler automatically distinguishes this instruction from the other RET pop-value instruction.

Bytes: 3
Clocks: 24, µPD70108
24, µPD70116 odd addresses
20, µPD70116 even addresses

Transfers: 1
Flag operation: None
Example: RET 8

RET (no operand)
Return from procedure, external segment

```
7 1 1 0 0 1 0 1 1
```

PC ← (SP + 1, SP),
PS ← (SP + 3, SP + 2),
SP ← SP + 4

Restores the PC and PS from the stack. Used for returning from intersegment calls. The assembler automatically distinguishes this instruction from the RET instruction without an operand.

Bytes: 1
Clocks: 29, µPD70108
29, µPD70116 odd addresses
21, µPD70116 even addresses

Transfers: 2
Flag operation: None
Example: RET
RET pop-value
Return from procedure, SP jump, intersegment

PC ← (SP + 1, SP),
PS ← (SP + 3, SP + 2),
SP ← SP + 4,
SP ← SP + pop-value

Restores the PC and PS from the stack and adds the
16-bit pop-value specified by the operand to the SP. This
command is effective for jumping the SP value when the
parameters saved in the stack subsequently become
unnecessary to the program. Used for returning from
intersegment calls. The assembler automatically distin-
guishes this instruction from the other RET pop-value
instruction.

Bytes: 3
Clocks:
32, μPD70108
32, μPD70116 odd addresses
24, μPD70116 even addresses

Transfers: 2
Flag operation: None
Example: RET 4

STACK OPERATION

PUSH mem16
Push, 16-bit memory

SP ← SP - 2

(SP - 1, SP - 2) ← (mem16),
SP ← SP - 2

Saves the contents of the 16-bit memory location
addressed by the operand to the stack.

Bytes: 2/3/4
Clocks:
26, μPD70108
26, μPD70116 odd addresses
18, μPD70116 even addresses

Transfers: 2
Flag operation: None
Example: PUSH DATA [IX]
PUSH reg16
Push, 16-bit register

\[
\begin{array}{ccccccc}
0 & 1 & 0 & 1 & 0 & \text{reg} & 0 \\
\end{array}
\]

\((SP - 1, SP - 2) \leftarrow \text{reg16},\)
\(SP \leftarrow SP - 2\)

Saves the 16-bit register specified by the operand to the stack.

Bytes: 1

Clocks:
- 12, \(\mu\)PD70108
- 12, \(\mu\)PD70116 odd addresses
- 8, \(\mu\)PD70116 even addresses

Transfers: 1

Flag operation: None

Example: PUSH IY

PUSH sreg
Push, segment register

\[
\begin{array}{ccccccc}
0 & 0 & 0 & sreg & 1 & 1 & 0 \\
\end{array}
\]

\((SP - 1, SP - 2) \leftarrow \text{sreg},\)
\(SP \leftarrow SP - 2\)

Saves the segment register specified by the operand to the stack.

Bytes: 1

Clocks:
- 12, \(\mu\)PD70108
- 12, \(\mu\)PD70116 odd addresses
- 8, \(\mu\)PD70116 even addresses

Transfers: 1

Flag operation: None

Example: PUSH PS
PUSH PSW

Push, program status word

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(SP - 1, SP - 2) ← PSW,
SP ← SP - 2
Saves the PSW to the stack.

Bytes: 1

Clocks:
12, µPD70108
12, µPD70116 odd addresses
8, µPD70116 even addresses

Transfers: 1

Flag operation: None

Example: PUSH PSW

---

PUSH R

Push, register set

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

temp ← SP,
(SP - 1, SP - 2) ← AW,
(SP - 3, SP - 4) ← CW,
(SP - 5, SP - 6) ← DW,
(SP - 7, SP - 8) ← BW,
(SP - 9, SP - 10) ← temp,
(SP - 11, SP - 12) ← BP,
(SP - 13, SP - 14) ← IX,
(SP - 15, SP - 16) ← IY,
SP ← SP - 16
Saves eight 16-bit registers (AW, BW, CW, DW, SP, BP, IX, and IY) to the stack.

Bytes: 1

Clocks:
67, µPD70108
67, µPD70116 odd addresses
35, µPD70116 even addresses

Transfers: 8

Flag operation: None

Example: PUSH R
### Section 12
**Instruction Set**

<table>
<thead>
<tr>
<th>PUSH imm8</th>
<th>PUSH imm16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push, 8-bit immediate data, sign expansion</td>
<td>Push, 16-bit immediate data</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(SP - 1, SP - 2) ← Sign expansion of imm8,</td>
<td>(SP - 1, SP - 2) ← imm16,</td>
</tr>
<tr>
<td>SP ← SP - 2</td>
<td>SP ← SP - 2</td>
</tr>
<tr>
<td>Expands the sign of the 8-bit immediate data specified</td>
<td>Saves the 16-bit immediate data described by the operand and to the stack addressed by the SP.</td>
</tr>
<tr>
<td></td>
<td>Bytes: 2</td>
</tr>
<tr>
<td>Clocks:</td>
<td>Bytes: 3</td>
</tr>
<tr>
<td>11, µPD70108</td>
<td>Clocks:</td>
</tr>
<tr>
<td>11, µPD70116 odd addresses</td>
<td>12, µPD70108</td>
</tr>
<tr>
<td>7, µPD70116 even addresses</td>
<td>12, µPD70116 odd addresses</td>
</tr>
<tr>
<td>Transfers: 1</td>
<td>8, µPD70116 even addresses</td>
</tr>
<tr>
<td>Flag operation: None</td>
<td>Transfers: 1</td>
</tr>
<tr>
<td>Example: PUSH 5</td>
<td>Flag operation: None</td>
</tr>
<tr>
<td>PUSH -1</td>
<td>Example: PUSH 1234H</td>
</tr>
</tbody>
</table>
POP mem16
Pop, 16-bit memory

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>mem</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(mem16) ← (SP + 1, SP),
SP ← SP + 2

Transfers the contents of the stack to the 16-bit memory location addressed by the operand.

Bytes: 2/3/4

Clocks:
- 25, μPD70108
- 25, μPD70116 odd addresses
- 17, μPD70116 even addresses

Transfers: 2
Flag operation: None
Example: POP DATA

POP reg16
Pop, 16-bit register

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

reg16 ← (SP + 1, SP), SP ← SP + 2

Transfers the contents of the stack to the 16-bit register specified by the operand.

Bytes: 1

Clocks:
- 12, μPD70108
- 12, μPD70116 odd addresses
- 8, μPD70116 even addresses

Transfers: 1
Flag operation: None
Example: POP BP
POP sreg

Pop, segment register

\[
\begin{array}{cccc}
7 & 6 & 5 & 0 \\
0 & 0 & 0 & \text{sreg} & 1 & 1 & 1
\end{array}
\]

sreg ← (SP + 1, SP), SP ← SP + 2

Transfers the contents of the stack to the segment register (except PS) specified by the operand. External interrupts NMI and INT, and single-step breaks will not be acknowledged between this instruction and the next.

Bytes: 1

Clocks:
12, \(\mu\)PD70108
12, \(\mu\)PD70116 odd addresses
8, \(\mu\)PD70116 even addresses

Transfers: 1

Flag operation: None

Example: POP DS1

POP PSW

Pop, program status word

\[
\begin{array}{cccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 0 & 1
\end{array}
\]

PSW ← (SP + 1, SP), SP ← SP + 2

Transfers the contents of the stack to the PSW.

Bytes: 1

Clocks:
12, \(\mu\)PD70108
12, \(\mu\)PD70116 odd addresses
8, \(\mu\)PD70116 even addresses

Transfers: 1

Flag operation:

\[
\begin{array}{cccccccc}
\text{MD}* & \text{V} & \text{DIR} & \text{IE} & \text{BRK} & \text{S} & \text{Z} \\
\text{R} & \text{R} & \text{R} & \text{R} & \text{R} & \text{R} & \text{R}
\end{array}
\]

\[
\begin{array}{cccc}
\text{AC} & \text{P} & \text{CY} \\
\text{R} & \text{R} & \text{R}
\end{array}
\]

*The Mode flag (MD) can only be modified by POP PSW during Native mode calls from 8080 Emulation mode; i.e. between the execution of BRKEM and RETEM instructions. In Native mode outside of Emulation mode, the MD flag will remain set to 1 regardless of the contents of the stack. Do not alter the MD flag during Native mode calls from Emulation mode, or during Native mode interrupt service routines which may be executed by interrupting Emulation mode execution.

Example: POP PSW
**POP R**

Pop, register set

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

IY ← (SP + 1, SP),
IX ← (SP + 3, SP + 2),
BP ← (SP + 5, SP + 4),
BW ← (SP + 9, SP + 8),
DW ← (SP + 11, SP + 10),
CW ← (SP + 13, SP + 12),
AW ← (SP + 15, SP + 14),
SP ← SP + 16

Restores the contents of the stack to the following 16-bit registers: AW, BW, CW, OW, BP, SP, IX, and IY.

Bytes: 1

Clocks:
75, μPD70108
75, μPD70116 odd addresses
43, μPD70116 even addresses

Transfers: 7

Flag operation: None

Example: POP R

**PREPARE imm16,imm8**

Prepare new stack frame

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(SP - 1, SP - 2) ← BP,
SP ← SP - 2,
temp ← SP,
When imm8 > 0, repeat these operations "imm8 - 1" times:
(SP - 1, SP - 2) ← (BP - 1, BP - 2)
SP ← SP - 2 (*1, see notes)
BP ← BP - 2
and perform these operations:
(SP - 1, SP - 2) ← temp
SP ← SP - 2 (*2, see notes)

Then perform these operations:
BP ← temp
SP ← SP - imm16

Notes: When imm8=1, *1 is not performed,
When imm8=0, *1 and *2 are not performed.

Used to generate "stack frames" required by the block structures of high-level languages such as Pascal and Ada. The stack frame includes a local variable area as well as pointers. These frame pointers point to other frames containing variables that can be referenced from the current procedure.

The first operand (16-bit immediate data) specifies (in bytes) the size of the local variable area. The second operand (8-bit immediate data) specifies the depth (or lexical level) of the procedure block. The frame base address generated by this instruction is set in the BP base pointer.

First the old BP value is saved to the stack so that BP of the calling procedure can be restored when the called procedure terminates. The frame pointer (BP value saved to the stack) that indicates the range of variables that can be referenced by the called procedure is placed on the stack. This range is always a value one less than the lexical level of the procedure. If the lexical level of a procedure is greater than one, the pointers of that procedure will also be saved on the stack. This enables the frame pointer of the calling procedure to be copied when frame pointer copy is performed within the called procedure.
Next, the new frame pointer value is set in the BP and the area for local variables used by the procedure is reserved in the stack. In other words, SP is decremented only for the amount of stack memory required by the local variables.

Bytes: 4

Clocks:
- When imm 8 = 0: 16, μPD70108
  - 16, μPD70116 odd addresses
  - 12, μPD70116 even addresses
- When 8 > 1: 23 + 16 (imm8 - 1), μPD70108
  - 23 + 16 (imm8 - 1), μPD70116
  - odd addresses
  - 19 + 8 (imm8 - 1), μPD70116
  - even addresses

Transfers:
- When imm8 = 0: none
- When imm8 > 1: 1 + 2(imm8-1)

Flag operation: None

Example: PREPARE 10, 3

---

**DISPOSE (no operand)**

Dispose a stack frame

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

SP ← BP,
BP ← (SP + 1, SP),
SP ← SP + 2

Releases the last stack frame generated by the PREPARE instruction. A value that points to the preceding frame is loaded in the BP and the bottom of the frame value is loaded in SP.

Bytes: 1

Clocks:
- 10, μPD70108
- 10, μPD70116 odd addresses
- 6 μPD70116 even addresses

Transfers: 1

Flag operation: None

Example: DISPOSE
BRANCH
BR-near-label
Branch Relative, Same Segment BR near-label
7 1 1 1 0 1 0 0 1
    (disp-low)
    (disp-high)
PC ← PC + disp
Loads the current PC value plus a 16-bit displacement value to the PC. If the branch address is in the current segment, the assembler automatically generates this instruction.
Bytes: 3
Clocks: 12
Transfers: None
Flag operation: None
Example: BR LABEL1

BR short-label
Branch short relative, same segment
7 1 1 1 0 1 0 1 1
    disp8
PC ← PC + ext-disp8
Loads the current PC value plus an 8-bit (actually, sign-extended 16-bit) displacement value to the PC. When the branch address is in the current segment and within ±127 bytes of the instruction, the assembler automatically generates this instruction.
Bytes: 2
Clocks: 12
Transfers: None
Flag operation: None
Example: BR SHORT_LABEL
BR regptr16
Branch register, same segment

```
  1 1 1 1 1 1 1 1
  1 1 1 0 0 reg
```

PC ← regptr16
Loads the contents of the 16-bit register specified by the operand to the PC. This instruction can branch to any address in the current segment.

Bytes: 2
Clocks: 11
Transfers: None
Flag operation: None
Example: BR BX

BR memptr16
Branch memory, same segment

```
  1 1 1 1 1 1 1 1
  1 1 1 0 0 mem
   (disp-low)
   (disp-high)
```

PC ← (memptr16)
Loads the contents of the 16-bit memory location addressed by the operand to the PC. This instruction can branch to any address in the current segment.

Bytes: 2/3/4
Clocks:
- 24, μPD70108
- 24, μPD70116 odd addresses
- 20, μPD70116 even addresses
Transfers: 1
Flag operation: None
Example: BR TABLE [IX]
BR far-label
Branch direct, external segment

\[
\begin{array}{ccccccc}
7 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

PC ← offset,
PS ← seg

Loads the 16-bit offset data (second and third bytes of the instruction) to the PC and the 16-bit segment data (fourth and fifth bytes) to the PS. This instruction can branch to any address in any segment.

Bytes: 5
Clocks: 15
Transfers: None
Flag operation: None
Example: BR FAR_LABEL

BR memptr32
Branch memory, external segment

\[
\begin{array}{ccccccc}
7 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

PS ← (memptr32 + 3, memptr32 + 2)
PC ← (memptr32 + 1, memptr32)

Loads the upper two bytes and lower two bytes of the 32-bit memory addressed by the operand to the PS and PC, respectively. This instruction can branch to any address in any segment.

Bytes: 2/3/4
Clocks: 35, \( \mu \text{PD70108} \)
35, \( \mu \text{PD70116} \) odd addresses
27, \( \mu \text{PD70116} \) even addresses
Transfers: 2
Flag operation: None
Example: BR FAR_SEGMENT [IY]
**CONDITIONAL BRANCH**

**BV short-label**

Branch if overflow

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

When $V = 1$, PC ← PC + ext-disp8

When the V flag is 1, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.

Bytes: 2

Clocks:
- When $V = 1$: 14
- When $V = 0$: 4

Transfers: None

Flag operation: None

Example: BV OVERFLOW_ERROR

**BNV short-label**

Branch if not overflow

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

When $V = 0$, PC ← PC + ext-disp8

When the V flag is 0, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.

Bytes: 2

Clocks:
- When $V = 0$: 14
- When $V = 1$: 4

Transfers: None

Flag operation: None

Example: BNV NO_ERROR
BC short-label  
BL short-label  
Branch if carry/lower  

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>disp8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When CY = 1, PC ← PC + ext-disp8  
When the CY flag is 1, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.  

Bytes: 2  
Clocks:  
When CY = 1: 14  
When CY = 0: 4  
Transfers: None  
Flag operation: None  
Example:  
BC CARRY_SET  
BL LESS_THAN

BNC short-label  
BNL short-label  
Branch if not carry/not lower  

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>disp8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When CY = 0, PC ← PC + ext-disp8  
When the CY flag is 0, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.  

Bytes: 2  
Clocks:  
When CY = 0: 14  
When CY = 1: 4  
Transfers: None  
Flag operation: None  
Example:  
BNC CARRY_CLEAR  
BNL GREATER_OR_EQUAL
### BE short-label
### BZ short-label

Branch if equal/zero

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

When \( Z = 1 \), \( PC \leftarrow PC + \text{ext-disp8} \)

When the Z flag is 1, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.

**Bytes:** 2

**Clocks:**
- When \( Z = 1 \): 14
- When \( Z = 0 \): 4

**Transfers:** None

**Flag operation:** None

**Example:**
- BE EQUALITY
- BZ ZERO

### BNE short-label
### BNZ short-label

Branch if not equal/not zero

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

When \( Z = 0 \), \( PC \leftarrow PC + \text{ext-disp8} \)

When the Z flag is 0, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.

**Bytes:** 2

**Clocks:**
- When \( Z = 0 \): 14
- When \( Z = 1 \): 4

**Transfers:** None

**Flag operation:** None

**Example:**
- BNE NOT_EQUAL
- BNZ NOT_ZERO
BNH short-label
Branch if not higher

When CY OR Z = 1, PC ← PC + ext-disp8
When the logical sum of the CY and Z flags is 1, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.

Bytes: 2
Clocks:
  When CY OR Z = 1: 14
  When CY OR Z = 0: 4
Transfers: None
Flag operation: None
Example: BNH NOT_HIGHER

BH short-label
Branch if higher

When CY OR Z = 0, PC ← PC + ext-disp8
When the logical sum of the CY and Z flags is 0, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.

Bytes: 2
Clocks:
  When CY OR Z = 0: 14
  When CY OR Z = 1: 4
Transfers: None
Flag operation: None
Example: BH HIGHER
**BN short-label**

Branch if negative

```
0 1 1 1 1 0 0 0
```

When $S = 1$, $PC \leftarrow PC + \text{ext-disp8}$

When the $S$ flag is 1, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.

Bytes: 2

Clocks:
- When $S = 1$: 14
- When $S = 0$: 4

Transfers: None

Flag operation: None

Example: BN NEGATIVE

---

**BP short-label**

Branch if positive

```
0 1 1 1 1 0 0 1
```

When $S = 0$, $PC \leftarrow PC + \text{ext-disp8}$

When the $S$ flag is 0, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.

Bytes: 2

Clocks:
- When $S = 0$: 14
- When $S = 1$: 4

Transfers: None

Flag operation: None

Example: BP POSITIVE
BPE short-label
Branch if parity even

When P = 1, PC ← PC + ext-disp8
When the P flag is 1, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.

Bytes: 2
Clocks:
  - When P = 1: 14
  - When P = 0: 4

Transfers: None
Flag operation: None
Example: BPE PARITY_EVEN

BPO short-label
Branch if parity odd

When P = 0, PC ← PC + ext-disp8
When the P flag is 0, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.

Bytes: 2
Clocks:
  - When P = 0: 14
  - When P = 1: 4

Transfers: None
Flag operation: None
Example: BPO PARITY_ODD
BLT short-label
Branch if less than

```
0 1 1 1 1 1 0 0
```

disp8

When S XOR V = 1, PC ← PC + ext-disp8
When the exclusive OR of the S and V flags is 1, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment. When the conditions are unsatisfied, proceeds to the next instruction.

Bytes: 2
Clocks:
  When S XOR V = 1: 14
  When S XOR V = 0: 4
Transfers: None
Flag operation: None
Example: BLT LESS_THAN

BGE short-label
Branch if greater than or equal

```
0 1 1 1 1 1 0 1
```

disp8

When S XOR V = 0, PC ← PC + ext-disp8
When the Exclusive OR of the S and V flags is 0, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment. When the conditions are unsatisfied, proceeds to the next instruction.

Bytes: 2
Clocks:
  When S XOR V = 0: 14
  When S XOR V = 1: 4
Transfers: None
Flag operation: None
Example: BGE GREATER_OR_EQUAL
BLE short-label
Branch if less than or equal

```
  7  0  1  1  1  1  1  1  0
```

When $(S \text{ XOR } V) \text{ OR } Z = 1$, $PC \leftarrow PC + \text{ext-disp8}$

When the Exclusive OR of the $S$ and $V$ flags and the logical sum of that result and the $Z$ flag is 1, loads the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment. When the conditions are unsatisfied, proceeds to the next instruction.

Bytes: 2
Clocks:
  When $(S \text{ XOR } V) \text{ OR } Z = 1$: 14
  When $(S \text{ XOR } V) \text{ OR } Z = 0$: 4

Transfers: None
Flag operation: None
Example: BLE LESS_OR_EQUAL

BGT short-label
Branch if greater than

```
  7  0  1  1  1  1  1  1  1
```

When $(S \text{ XOR } V) \text{ OR } Z = 0$, $PC \leftarrow PC + \text{ext-disp8}$

When the exclusive OR of the $S$ and $V$ flags and the logical sum of that result and the $Z$ flag is 0, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment. When the conditions are unsatisfied, proceeds to the next instruction.

Bytes: 2
Clocks:
  When $(S \text{ XOR } V) \text{ OR } Z = 0$: 14
  When $(S \text{ XOR } V) \text{ OR } Z = 1$: 4

Transfers: None
Flag operation: None
Example: BGT GREATER
DBNZNE short-label
Decrement and branch if not zero and not equal

\[
\begin{array}{cccccc}
7 & 1 & 1 & 1 & 0 & 0 \\
\hline
\text{disp8} & 0 & 0 & 0 & 0 & 0
\end{array}
\]

When the 16-bit register \( CW \) is decremented \((-1)\), the resultant \( CW \) value is not 0, and the \( Z \) flag is cleared, load the current \( PC \) value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the \( PC \). This instruction can branch to any address within ±127 bytes of the instruction in the current segment.

Bytes: 2
Clocks:
  - When \( CW \neq 0 \) and \( Z = 0 \): 14
  - When others: 5

Transfers: None
Flag operation: None
Example: PBNZNE LOOP_AGAIN

DBNZE short-label
Decrement and branch if not zero and equal

\[
\begin{array}{cccccc}
7 & 1 & 1 & 1 & 0 & 0 \\
\hline
\text{disp8} & 0 & 0 & 0 & 0 & 1
\end{array}
\]

When the 16-bit register \( CW \) is decremented \((-1)\), the \( CW \) is not zero, and the \( Z \) flag is set, load the current \( PC \) value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the \( PC \). This instruction can branch to any address within ±127 bytes of the instruction in the current segment.

Bytes: 2
Clocks:
  - When \( CW \neq 0 \) and \( Z = 1 \): 14
  - When others: 5

Transfers: None
Flag operation: None
Example: DBNZE LOOP_AGAIN
**DBNZ short-label**

Decrement and branch if not zero

```
7 0
1 1 1 0 0 0 1 0
```

**Example:**

```
DBNZ LOOP_AGAIN
```

**BCWZ short-label**

Branch if CW equals zero

```
7 0
1 1 1 0 0 0 1 1
```

**Example:**

```
BCWZ CW_ZERO
```

**DBNZ short-label**

- **Operation:** CW ← CW − 1
  - When CW ≠ 0, PC ← PC + ext-disp8
- **Description:** When the 16-bit register CW is decremented (−1) and the CW value is not zero, load the current PC value plus the 8-bit (actually, sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.
- **Bytes:** 2
- **Clocks:**
  - When CW ≠ 0: 13
  - When CW = 0: 5
- **Transfers:** None
- **Flag operation:** None

**BCWZ short-label**

- **Operation:** If CW = 0, PC ← PC + ext-disp8
- **Description:** When the 16-bit register CW is 0, load the current PC value plus the 8-bit (actually sign-extended 16-bit) displacement value to the PC. This instruction can branch to any address within ±127 bytes of the instruction in the current segment.
- **Bytes:** 2
- **Clocks:**
  - When CW = 0: 13
  - When CW ≠ 0: 5
- **Transfers:** None
- **Flag operation:** None

**Example:**

```
DBNZ LOOP_AGAIN
```

```
BCWZ CW_ZERO
```
BREAK

BRK 3

Break, vector 3

\[
\begin{array}{cccccc}
7 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\((SP - 1, SP - 2) \rightarrow PSW\)
\((SP - 3, SP - 4) \rightarrow PS\)
\((SP - 5, SP - 6) \rightarrow PC\)
\(SP \rightarrow SP - 6\)
\(IE \leftarrow 0\)
\(BRK \leftarrow 0\)
\(PC \leftarrow (13, 12)\)
\(PS \leftarrow (15, 14)\)

Saves the PSW, PS, and PC to the stack and resets the IE and BRK flags to 0. Then loads the lower two bytes and higher two bytes of vector 3 of the interrupt vector table to the PC and PS, respectively.

Bytes: 1

Clocks:
50, \(\mu\)PD70108
50, \(\mu\)PD70116 odd addresses
38, \(\mu\)PD70116 even addresses

Transfers: 5

Flag operation:

<table>
<thead>
<tr>
<th>IE</th>
<th>BRK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: BRK 3

BRK imm8 (\(\neq 3\))

Break, immediate data

\[
\begin{array}{cccccc}
7 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
\end{array}
\]

\((SP - 1, SP - 2) \rightarrow PSW\)
\((SP - 3, SP - 4) \rightarrow PS\)
\((SP - 5, SP - 6) \rightarrow PC\)
\(SP \rightarrow SP - 6\)
\(IE \leftarrow 0\)
\(BRK \leftarrow 0\)
\(PC \leftarrow (imm8 \times 4 + 1, imm8 \times 4)\)
\(PS \leftarrow (imm8 \times 4 + 3, imm8 \times 4 + 2)\)

Saves the PSW, PS, and PC to the stack and resets the IE and BRK flags to 0. Then loads the lower two bytes and upper two bytes of the interrupt vector table (4 bytes) specified by the 8-bit immediate data to the PC and PS, respectively.

Bytes: 1

Clocks:
50, \(\mu\)PD70108
50, \(\mu\)PD70116 odd addresses
38, \(\mu\)PD70116 even addresses

Transfers: 5

Flag operation:

<table>
<thead>
<tr>
<th>IE</th>
<th>BRK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: BRK 10H ;PC = (40H,41H), ;PS = (42H,43H)
BRKV (no operand)

Break if overflow

```
| 7 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
```

When \( V = 1 \),
(\( SP - 1\), \( SP - 2\)) \( \leftarrow \) PSW
(\( SP - 3\), \( SP - 4\)) \( \leftarrow \) PS
(\( SP - 5\), \( SP - 6\)) \( \leftarrow \) PC
\( SP \leftarrow SP - 6\)
\( IE \leftarrow 0\)
\( BRK \leftarrow 0\)
\( PC \leftarrow (011H, 010H)\)
\( PS \leftarrow (013H, 012H)\)

When the \( V \) flag is set, saves the PSW, PS, and PC to the stack and resets the IE and BRK flags to 0. Then loads the lower two bytes and upper two bytes of vector 4 of the interrupt vector table to the PC and PS, respectively. When the \( V \) flag is reset, proceeds to the next instruction.

Bytes: 1

Clocks:
When \( V = 1 \):
52, \( \mu \)PD70108
52, \( \mu \)PD70116 odd addresses
When \( V = 0 \):
40, \( \mu \)PD70116 even addresses

Transfers: 5

Flag operation:

<table>
<thead>
<tr>
<th>IE</th>
<th>BRK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: BRKV

RETI (no operand)

Return from interrupt

```
| 7 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
```

\( PC \leftarrow (SP + 1\), SP\)
\( PS \leftarrow (SP + 3\), \( SP + 2\)\)
\( PSW \leftarrow (SP + 5\), \( SP + 4\)\)
\( SP \leftarrow SP + 6\)

Restores the contents of the stack to the PC, PS, and PSW. Used for return from interrupt processing.

Bytes: 1

Clocks:
39, \( \mu \)PD70108
39, \( \mu \)PD70116 odd addresses
27, \( \mu \)PD70116 even addresses

Transfers: 3

Flag operation:

<table>
<thead>
<tr>
<th>MD*</th>
<th>V</th>
<th>DIR</th>
<th>IE</th>
<th>BRK</th>
<th>S</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

*The Mode flag (MD) can only be modified by RETI during Native mode calls from 8080 Emulation mode; i.e. between the execution of BRKEM and RETEM instructions. In Native mode outside of Emulation mode, the MD flag will remain set to 1 regardless of the contents of the stack. Do not alter the MD flag during Native mode calls from Emulation mode, or during interrupt service routines which may be executed by interrupting Emulation mode execution. The RETI instruction should be used to exit Native mode service routines and to return to Emulation mode. The RETI instruction should be the last instruction executed in the Native mode service routine.

Example: RETI
BRKEM imm8

Break for emulation

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(SP - 1, SP - 2) ← PSW
(SP - 3, SP - 4) ← PS
(SP - 5, SP - 6) ← PC
SP ← SP - 6
MD ← 0, write enable MD
PS ← (imm 8 × 4 + 3, imm 8 × 4 + 2)
PC ← imm 8 × 4 + 1, imm 8 × 4)

Starts the emulation mode. Saves the PSW, PS, and PC and resets the MD bit to 0 and jumps to the emulation location addressed by the interrupt vector specified by the 8-bit immediate data specified by the operand. After fetching the instruction code of the jumped interrupt service routine (for emulation), the CPU interprets and executes the code as an instruction of the μPD808AF. Use either the RETEM or CALLN instruction to return from the emulation mode to the native mode (μPD70108/70116). CALLN temporarily returns the program from Emulation to Native Mode and RETEM completes Emulation mode.

Bytes: 3
Clocks:
50, μPD70108
50, μPD70116 odd addresses
38, μPD70116 even addresses

Transfers: 5
Flag operation:

<table>
<thead>
<tr>
<th>MD</th>
<th>0</th>
</tr>
</thead>
</table>

Example: BRKEM 80H
CHKIND reg16, mem32

Check index

<table>
<thead>
<tr>
<th>Mod</th>
<th>Reg</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

When (mem32) > reg16 or (mem32 + 2) < reg16
(SP - 1, SP - 2) ← PSW
(SP - 3, SP - 4) ← PS
(SP - 5, SP - 6) ← PC
SP ← SP - 6
IE ← 0
BRK ← 0
PS ← (23, 22)
PC ← (21, 20)

Used to check whether the index value in reg16 is within the defined array bounds. Initiates a BRK 5 when the index does not satisfy the condition. The definition region should be set beforehand in the two words (first word for the lower limit and second word for the upper limit) of memory.

Bytes: 2/3/4

Clocks:
When interrupt condition is fulfilled:
73-76, \( \mu \text{PD70108} \)
73-76, \( \mu \text{PD70116} \) odd addresses
53-56, \( \mu \text{PD70116} \) even addresses
When interrupt condition is not fulfilled:
26, \( \mu \text{PD70108} \)
26, \( \mu \text{PD70116} \) odd addresses
18, \( \mu \text{PD70116} \) even addresses

Transfers:
When interrupt condition is fulfilled: 7
When interrupt condition is not fulfilled: 2

Flag operation:
When interrupt condition is fulfilled:

<table>
<thead>
<tr>
<th>IE</th>
<th>BRK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example:
When interrupt condition is not fulfilled: None:

Example:
MOV IX, 23
CHKIND IX, BOUNDS1 ; OK
MOV BW, 87
CHKIND BW, BOUNDS2 ; causes ; BRK 5

BOUNDSS1 DW 5, 37
BOUNDSS2 DW 2, 80
**CPU CONTROL**

**HALT (no operand)**

Halt

```
  7  1  1  1  1  0  1  0  0
```

Sets the halt state. The halt state is released by the RESET, NMI, or INT input.

Bytes: 1
Clocks: 2
Transfers: None
Flag operation: None
Example: HALT

**POLL (no operand)**

Poll and wait

```
  1  0  0  1  1  0  1  1  0
```

Keeps the CPU in the idle state until the POLL pin becomes an active low level.

Bytes: 1
Clocks: $2 + 5n$, where $n =$ number of times POLL pin is sampled
Transfers: None
Flag operation: None
Example: POLL
**DI (no operand)**

Disable interrupt

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

IE ← 0

Resets the IE flag and disables the external maskable interrupt input (INT). Does not disable the external non-maskable interrupt input (NMI) or software interrupt instructions.

Bytes: 1
Clocks: 2
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>IE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Example: DI

---

**EI (no operand)**

Enable interrupt

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

EI ← 1

Sets the EI flag and enables the external maskable interrupt input (INT). The system does not enter the interrupt-enable state until executing the instruction immediately after EI.

Bytes: 1
Clocks: 2
Transfers: None
Flag operation:

<table>
<thead>
<tr>
<th>IE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Example: EI
BUSLOCK (no operand)

Bus lock prefix

| 7 | 1 1 1 1 0 0 0 0 |

In the large-scale mode (S/LG = 0)

Outputs the buslock signal (BUSLOCK) while the instruction immediately after the BUSLOCK instruction is being executed. When BUSLOCK is used for a block operation instruction with a repeat prefix, the BUSLOCK signal is kept at an active low level until the end of the block operation instruction.

Hold request is inhibited when BUSLOCK is active. The BUSLOCK instruction is effective when you do not want to acknowledge a hold request during block operations.

In small-scale mode (S/LG = 1)

The BUSLOCK signal is not an output. However, the BUSLOCK instruction can be used to delay a hold acknowledge response to a hold request until execution of the locked instruction is completed.

Bytes: 1
Clocks: 2
Transfers: None
Flag operation: None
Example: BUSLOCK REP MOVKBK

FPO1 fp-op

Floating point operation 1, register

| 7 | 1 1 0 1 1 X X X |

| 1 1 Y Y Y Z Z Z |

Used when the floating point arithmetic chip is connected externally. Causes the CPU to leave arithmetic processing to the floating point chip. When the floating point chip monitors this instruction, it treats the instruction as its own and executes it.

Bytes: 2
Clocks: 2
Transfers: None
Flag operation: None
Example: FPO1 FABS0
FPO1 FCMPR2
Floating point operation 1, memory

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>X</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
</table>

**mod** | Y | Y | Y | **mem**

**(disp-low)**

**(disp-high)**

Data bus ← (mem)

Used when the floating point arithmetic chip is externally connected. Causes the CPU to leave arithmetic processing to the floating point chip and instead, carries out auxiliary processing such as calculation of effective address, generation of physical addresses, and start of memory read cycles when necessary.

When the floating point chip monitors this instruction, it treats the instruction as its own and executes it. In this case, depending on the type of instruction, the floating point chip selects either the address information of the memory read cycle started by the CPU or both the address and read data. The CPU does not use the read data on the data bus in the memory read cycle which the CPU has initiated.

**Bytes:** 2/3/4

**Clocks:** 15, \(\mu\)PD70108
15, \(\mu\)PD70116 odd addresses
11, \(\mu\)PD70116 even addresses

**Transfers:** 1

**Flag operation:** None

Example:
FPO1 FCMP,DWORD_VAR
FPO1 FMUL,QWORD PTR [BW]

Floating point operation 2, register

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>X</th>
</tr>
</thead>
</table>

0 | 1 | 1 | 0 | 0 | 1 | 1 | X |

1 | 1 | Y | Y | Y | Z | Z | Z |

Used with an externally connected floating point arithmetic chip. Causes the CPU to leave processing to the floating point chip. When the floating point chip monitors this instruction, it interprets the instruction as its own and executes it.

**Bytes:** 2

**Clocks:** 2

**Transfers:** None

**Flag operation:** None

Example: FPO2 FSINR0
FPO2 fp-op, mem
Floating point operation 2, memory

```
0 1 1 0 0 1 1 X
mod Y Y Y mem
(disp-low)
(disp-high)
```

Data bus ← (mem)

Used with an externally connected floating point arithmetic chip. Causes the CPU to leave arithmetic processing to the floating point chip and instead carries out auxiliary processing such as calculation of effective addresses, generation of physical addresses, and start of memory read cycles when necessary.

When the floating point chip monitors this instruction, it treats the instruction as its own and executes it. In this case, depending on the type of instruction, the operating chip selects either the address information of the memory read cycle started by the CPU or both the address and read data.

Bytes: 2/3/4
Clocks: 15, $\mu$PD70108
15, $\mu$PD70116 odd addresses
11, $\mu$PD70116 even addresses
Transfers: 1
Flag operation: None
Example: FPO2 FCOS,DWORD PTR [IX][BW]

NOP (no operand)
No operation

```
0 0 0 1 0 0 0 0
```

Causes the processor to do nothing for three clocks.

Bytes: 1
Clocks: 3
Transfers: None
Flag operation: None
Example: NOP
SEGMENT OVERRIDE PREFIXES

When appended to the operand, specifies the segment register to be used for access of a memory operand expecting segment override.

You can define the segment override by assembler directive "ASSUME" without describing the segment override prefix directly (see Assembler Operating Manual).

Example:

<table>
<thead>
<tr>
<th>MOV</th>
<th>IX,DS1:[IY]</th>
</tr>
</thead>
<tbody>
<tr>
<td>REP</td>
<td>MOV8KB DEST_BLK,SS:SRC_BLK</td>
</tr>
</tbody>
</table>

EMULATION MODE

CALLN imm8

Call native

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

When executed in the emulation mode, the CPU interprets the instruction as a μPD8080AF command. The CPU saves the PS, PC, and PSW to the stack (MD = 0 is also saved). Then the MD flag is set to 1. The interrupt vector specified by the 8-bit immediate data of the operand is loaded into PS and PC. This command allows you to call a native mode interrupt routine from the emulation mode.

The RETI command is used to return to emulation mode from the interrupt routine.

Example: CALLN 40H
RETEM
Return from emulation

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

PC ← (SP + 1, SP)
PS ← (SP + 3, SP + 2)
PSW ← (SP + 5, SP + 4)
SP ← SP + 6
Write disable MD

When executed in the emulation mode, the CPU interprets this instruction as a μPD8080AF command. The CPU restores the PS, PC, and PSW saved by the BRKEM command in the same manner as when returning from interrupt processing. When the BRKEM instruction is executed, the MD flag is write disabled, so the MD flag is not restored by executing the RETI or POP PSW instructions.

Bytes: 2
Clocks:
- 39, μPD70108
- 39, μPD70116 odd addresses
- 27, μPD70116 even addresses

Transfers: 3
Flag operation:

<table>
<thead>
<tr>
<th>MD</th>
<th>V</th>
<th>DIR</th>
<th>IE</th>
<th>BRK</th>
<th>S</th>
<th>Z</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

Example: RETEM
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>12-29</td>
<td>CALL</td>
<td>12-142</td>
</tr>
<tr>
<td>mem,reg</td>
<td>12-30</td>
<td>regptr16</td>
<td>12-142</td>
</tr>
<tr>
<td>reg,mem</td>
<td>12-30</td>
<td>memptr16</td>
<td>12-143</td>
</tr>
<tr>
<td>reg,imm</td>
<td>12-31</td>
<td>far-proc</td>
<td>12-143</td>
</tr>
<tr>
<td>mem,imm</td>
<td>12-31</td>
<td>memptr32</td>
<td>12-144</td>
</tr>
<tr>
<td>acc,imm</td>
<td>12-32</td>
<td>CALLN</td>
<td>12-176</td>
</tr>
<tr>
<td></td>
<td></td>
<td>imm8</td>
<td>12-176</td>
</tr>
<tr>
<td>ADDC</td>
<td>12-32</td>
<td>CHKIND</td>
<td>12-170</td>
</tr>
<tr>
<td>reg,reg</td>
<td>12-32</td>
<td>reg16,mem32</td>
<td>12-170</td>
</tr>
<tr>
<td>mem,reg</td>
<td>12-33</td>
<td>reg8,CL</td>
<td>12-90</td>
</tr>
<tr>
<td>reg,mem</td>
<td>12-33</td>
<td>mem8,CL</td>
<td>12-90</td>
</tr>
<tr>
<td>reg,imm</td>
<td>12-34</td>
<td>reg16,CL</td>
<td>12-91</td>
</tr>
<tr>
<td>mem,imm</td>
<td>12-34</td>
<td>mem16,CL</td>
<td>12-91</td>
</tr>
<tr>
<td>acc,imm</td>
<td>12-35</td>
<td>reg8,imm3</td>
<td>12-92</td>
</tr>
<tr>
<td>ADD4S</td>
<td>12-42</td>
<td>mem8,imm3</td>
<td>12-92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reg16,imm4</td>
<td>12-93</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem16,imm4</td>
<td>12-93</td>
</tr>
<tr>
<td>ADJBA</td>
<td>12-61</td>
<td>CY</td>
<td>12-94</td>
</tr>
<tr>
<td>ADJBS</td>
<td>12-62</td>
<td>DIR</td>
<td>12-94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem8,CL</td>
<td>12-92</td>
</tr>
<tr>
<td>ADD4A</td>
<td>12-61</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>12-72</td>
<td>CMP</td>
<td>12-65</td>
</tr>
<tr>
<td>reg,reg</td>
<td>12-72</td>
<td>mem,reg</td>
<td>12-65</td>
</tr>
<tr>
<td>mem,reg</td>
<td>12-72</td>
<td>reg,mem</td>
<td>12-66</td>
</tr>
<tr>
<td>reg,mem</td>
<td>12-73</td>
<td>reg,imm</td>
<td>12-66</td>
</tr>
<tr>
<td>reg,imm</td>
<td>12-74</td>
<td>mem,imm</td>
<td>12-67</td>
</tr>
<tr>
<td>acc,imm</td>
<td>12-75</td>
<td>acc,imm</td>
<td>12-67</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>12-158</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCWZ</td>
<td>12-166</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BE</td>
<td>12-166</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BGE</td>
<td>12-163</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BGT</td>
<td>12-164</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BH</td>
<td>12-164</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BL</td>
<td>12-160</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLE</td>
<td>12-164</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLT</td>
<td>12-163</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BN</td>
<td>12-161</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNC</td>
<td>12-158</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>12-159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNH</td>
<td>12-160</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNL</td>
<td>12-158</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNV</td>
<td>12-157</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNZ</td>
<td>12-159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td>12-161</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPE</td>
<td>12-162</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPO</td>
<td>12-162</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td>12-154</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRK</td>
<td>12-167</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRKEM</td>
<td>12-169</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRKV</td>
<td>12-168</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUSLOCK</td>
<td>12-173</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BV</td>
<td>12-157</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BZ</td>
<td>12-159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Page</td>
<td>Instruction</td>
<td>Page</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>FPO1</td>
<td>12-173</td>
<td>NOT1</td>
<td>12-85</td>
</tr>
<tr>
<td>fp-op</td>
<td>12-174</td>
<td>reg8,CL</td>
<td>12-86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem8,CL</td>
<td>12-86</td>
</tr>
<tr>
<td>FPO2</td>
<td>12-174</td>
<td>reg16,CL</td>
<td>12-87</td>
</tr>
<tr>
<td>fp-op</td>
<td>12-175</td>
<td>mem16,CL</td>
<td>12-87</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reg8,imm3</td>
<td>12-87</td>
</tr>
<tr>
<td>HALT</td>
<td>12-171</td>
<td>mem8,imm3</td>
<td>12-88</td>
</tr>
<tr>
<td>IN</td>
<td>12-25</td>
<td>reg16,imm4</td>
<td>12-88</td>
</tr>
<tr>
<td>acc,imm8</td>
<td></td>
<td>mem16,imm4</td>
<td>12-89</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem,imm</td>
<td>12-89</td>
</tr>
<tr>
<td>INC</td>
<td>12-47</td>
<td>CY</td>
<td></td>
</tr>
<tr>
<td>reg8</td>
<td></td>
<td>OR</td>
<td>12-75</td>
</tr>
<tr>
<td>mem</td>
<td>12-47</td>
<td>reg,reg</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem,reg</td>
<td>12-76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reg,mem</td>
<td>12-76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reg,imm</td>
<td>12-77</td>
</tr>
<tr>
<td>INM</td>
<td>12-28</td>
<td>mem,imm</td>
<td>12-77</td>
</tr>
<tr>
<td>dst-block,DW</td>
<td></td>
<td>acc,imm</td>
<td>12-78</td>
</tr>
<tr>
<td>INS</td>
<td>12-21</td>
<td>OUT</td>
<td>12-26</td>
</tr>
<tr>
<td>reg18,reg28</td>
<td></td>
<td>imm8,acc</td>
<td></td>
</tr>
<tr>
<td>reg8,imm4</td>
<td>12-22</td>
<td>DW,acc</td>
<td>12-26</td>
</tr>
<tr>
<td>LDEA</td>
<td>12-11</td>
<td>OUTM</td>
<td>12-29</td>
</tr>
<tr>
<td>reg16,mem16</td>
<td></td>
<td>DW,src-block</td>
<td></td>
</tr>
<tr>
<td>LDM</td>
<td>12-19</td>
<td>POLL</td>
<td></td>
</tr>
<tr>
<td>src-block</td>
<td></td>
<td>POP</td>
<td>12-171</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem16</td>
<td>12-150</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reg16</td>
<td>12-150</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sreg</td>
<td>12-151</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSW</td>
<td>12-151</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td>12-152</td>
</tr>
<tr>
<td>MOV</td>
<td>12-4</td>
<td>PREPARE</td>
<td>12-152</td>
</tr>
<tr>
<td>reg,reg</td>
<td></td>
<td>imm16,imm8</td>
<td></td>
</tr>
<tr>
<td>mem,reg</td>
<td></td>
<td>PS:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>imm8</td>
<td>12-176</td>
</tr>
<tr>
<td></td>
<td></td>
<td>imm16</td>
<td>12-149</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem16</td>
<td>12-146</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reg16</td>
<td>12-146</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sreg</td>
<td>12-147</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSW</td>
<td>12-148</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td>12-148</td>
</tr>
<tr>
<td>MOVBK</td>
<td>12-16</td>
<td>REP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>REP</td>
<td>12-15</td>
</tr>
<tr>
<td>MOVBKB</td>
<td>12-16</td>
<td>REPE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>REPNC</td>
<td>12-15</td>
</tr>
<tr>
<td>MOVBKW</td>
<td>12-16</td>
<td>REPNE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>REPNZ</td>
<td>12-15</td>
</tr>
<tr>
<td>MUL</td>
<td>12-52</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>reg8</td>
<td></td>
<td>RETM</td>
<td>12-144</td>
</tr>
<tr>
<td>mem8</td>
<td></td>
<td>RET1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pop-value</td>
<td>12-145</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROL</td>
<td>12-177</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reg,1</td>
<td>12-168</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem,1</td>
<td>12-118</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reg,CL</td>
<td>12-119</td>
</tr>
<tr>
<td>NEG</td>
<td>12-69</td>
<td>mem,CL</td>
<td>12-120</td>
</tr>
<tr>
<td>reg</td>
<td></td>
<td>reg,imm8</td>
<td>12-121</td>
</tr>
<tr>
<td>mem</td>
<td>12-69</td>
<td>mem,imm8</td>
<td>12-122</td>
</tr>
</tbody>
</table>

A-2
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page 1</th>
<th>Instruction</th>
<th>Page 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLC</td>
<td>12-130</td>
<td>SS:</td>
<td>12-176</td>
</tr>
<tr>
<td>reg,1</td>
<td>12-130</td>
<td>dst-block</td>
<td>12-20</td>
</tr>
<tr>
<td>mem,1</td>
<td>12-131</td>
<td>STM</td>
<td>12-19</td>
</tr>
<tr>
<td>reg,CL</td>
<td>12-132</td>
<td>STMB</td>
<td>12-19</td>
</tr>
<tr>
<td>mem,CL</td>
<td>12-133</td>
<td>STMW</td>
<td>12-19</td>
</tr>
<tr>
<td>reg,imm8</td>
<td>12-134</td>
<td>SUB</td>
<td>12-35</td>
</tr>
<tr>
<td>mem,imm8</td>
<td>12-135</td>
<td>reg,reg</td>
<td>12-36</td>
</tr>
<tr>
<td>ROL4</td>
<td>12-45</td>
<td>mem,reg</td>
<td>12-36</td>
</tr>
<tr>
<td>mem8</td>
<td>12-45</td>
<td>reg,imm</td>
<td>12-37</td>
</tr>
<tr>
<td>reg8</td>
<td>12-45</td>
<td>mem,reg</td>
<td>12-37</td>
</tr>
<tr>
<td>ROR</td>
<td>12-124</td>
<td>mem,imm</td>
<td>12-37</td>
</tr>
<tr>
<td>reg,1</td>
<td>12-124</td>
<td>SUBC</td>
<td>12-38</td>
</tr>
<tr>
<td>mem,1</td>
<td>12-125</td>
<td>mem,reg</td>
<td>12-38</td>
</tr>
<tr>
<td>reg,CL</td>
<td>12-126</td>
<td>reg,imm</td>
<td>12-39</td>
</tr>
<tr>
<td>mem,CL</td>
<td>12-127</td>
<td>mem,reg</td>
<td>12-39</td>
</tr>
<tr>
<td>reg,imm8</td>
<td>12-128</td>
<td>reg,imm</td>
<td>12-40</td>
</tr>
<tr>
<td>mem,imm8</td>
<td>12-129</td>
<td>mem,imm</td>
<td>12-40</td>
</tr>
<tr>
<td>RORC</td>
<td>12-136</td>
<td>acc,imm</td>
<td>12-41</td>
</tr>
<tr>
<td>reg,1</td>
<td>12-136</td>
<td>SUB4S</td>
<td>12-43</td>
</tr>
<tr>
<td>mem,1</td>
<td>12-137</td>
<td>TEST</td>
<td>12-70</td>
</tr>
<tr>
<td>reg,CL</td>
<td>12-138</td>
<td>reg,reg</td>
<td>12-70</td>
</tr>
<tr>
<td>mem,CL</td>
<td>12-139</td>
<td>mem,reg</td>
<td>12-70</td>
</tr>
<tr>
<td>reg,imm8</td>
<td>12-140</td>
<td>reg,imm</td>
<td>12-71</td>
</tr>
<tr>
<td>mem,imm8</td>
<td>12-141</td>
<td>mem,imm</td>
<td>12-71</td>
</tr>
<tr>
<td>ROR4</td>
<td>12-46</td>
<td>acc,imm</td>
<td>12-72</td>
</tr>
<tr>
<td>reg8</td>
<td>12-95</td>
<td>mem8</td>
<td>12-72</td>
</tr>
<tr>
<td>mem8</td>
<td>12-95</td>
<td>mem8,CL</td>
<td>12-82</td>
</tr>
<tr>
<td>SET1</td>
<td>12-95</td>
<td>mem16,CL</td>
<td>12-82</td>
</tr>
<tr>
<td>reg8,CL</td>
<td>12-95</td>
<td>reg16,CL</td>
<td>12-82</td>
</tr>
<tr>
<td>mem8,CL</td>
<td>12-95</td>
<td>mem16,CL</td>
<td>12-83</td>
</tr>
<tr>
<td>reg16,CL</td>
<td>12-96</td>
<td>reg8,imm3</td>
<td>12-83</td>
</tr>
<tr>
<td>mem16,CL</td>
<td>12-96</td>
<td>mem8,imm3</td>
<td>12-84</td>
</tr>
<tr>
<td>reg8,imm3</td>
<td>12-96</td>
<td>mem16,imm4</td>
<td>12-84</td>
</tr>
<tr>
<td>mem8,imm3</td>
<td>12-97</td>
<td>reg16,imm4</td>
<td>12-85</td>
</tr>
<tr>
<td>reg16,imm4</td>
<td>12-98</td>
<td>mem16,imm4</td>
<td>12-85</td>
</tr>
<tr>
<td>mem16,imm4</td>
<td>12-98</td>
<td>CY</td>
<td>12-99</td>
</tr>
<tr>
<td>SHL</td>
<td>12-100</td>
<td>TRANS</td>
<td>12-12</td>
</tr>
<tr>
<td>reg,1</td>
<td>12-100</td>
<td>TRANSB</td>
<td>12-12</td>
</tr>
<tr>
<td>mem,1</td>
<td>12-101</td>
<td>XCH</td>
<td>12-12</td>
</tr>
<tr>
<td>reg,CL</td>
<td>12-102</td>
<td>reg,reg</td>
<td>12-12</td>
</tr>
<tr>
<td>mem,CL</td>
<td>12-103</td>
<td>mem,reg</td>
<td>12-13</td>
</tr>
<tr>
<td>reg,imm8</td>
<td>12-104</td>
<td>AW,reg16</td>
<td>12-13</td>
</tr>
<tr>
<td>mem,imm8</td>
<td>12-105</td>
<td>XOR</td>
<td>12-78</td>
</tr>
<tr>
<td>SHR</td>
<td>12-106</td>
<td>reg,reg</td>
<td>12-78</td>
</tr>
<tr>
<td>reg,1</td>
<td>12-106</td>
<td>mem,reg</td>
<td>12-79</td>
</tr>
<tr>
<td>mem,1</td>
<td>12-107</td>
<td>reg,imm</td>
<td>12-79</td>
</tr>
<tr>
<td>reg,CL</td>
<td>12-108</td>
<td>mem,imm</td>
<td>12-80</td>
</tr>
<tr>
<td>mem,CL</td>
<td>12-109</td>
<td>acc,imm</td>
<td>12-81</td>
</tr>
<tr>
<td>SHR4</td>
<td>12-110</td>
<td>mem,imm</td>
<td>12-80</td>
</tr>
<tr>
<td>reg,1</td>
<td>12-110</td>
<td>src-table</td>
<td>12-12</td>
</tr>
<tr>
<td>mem,1</td>
<td>12-111</td>
<td>mem,imm</td>
<td>12-81</td>
</tr>
<tr>
<td>reg,CL</td>
<td>12-112</td>
<td>mem,imm</td>
<td>12-81</td>
</tr>
<tr>
<td>mem,CL</td>
<td>12-113</td>
<td>mem,imm</td>
<td>12-81</td>
</tr>
<tr>
<td>reg,imm8</td>
<td>12-114</td>
<td>mem,imm</td>
<td>12-81</td>
</tr>
<tr>
<td>mem,imm8</td>
<td>12-115</td>
<td>mem,imm</td>
<td>12-81</td>
</tr>
<tr>
<td>SHRA</td>
<td>12-116</td>
<td>mem,imm</td>
<td>12-81</td>
</tr>
<tr>
<td>reg,1</td>
<td>12-116</td>
<td>mem,imm</td>
<td>12-81</td>
</tr>
<tr>
<td>mem,1</td>
<td>12-117</td>
<td>mem,imm</td>
<td>12-81</td>
</tr>
</tbody>
</table>