Mainframe Assembler Mini-Reference

Instruction Formats

Note: In the following, only some 32-bit unprivileged instructions of zArchitecture are described. So register refers to low 32-bit portions of 64-bit general zArchitecture registers. Also, certain special cases are omitted for simplicity.

<table>
<thead>
<tr>
<th>Byte Fmt</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Op</td>
<td>I1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>Op</td>
<td>R1</td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RI</td>
<td>Op</td>
<td>R1</td>
<td>Op</td>
<td>I2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS</td>
<td>Op</td>
<td>R1</td>
<td>R2</td>
<td>B2</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>RX</td>
<td>Op</td>
<td>R1</td>
<td>M1</td>
<td>B2</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Op</td>
<td>B1</td>
<td>D1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>Op</td>
<td>L1</td>
<td>B1</td>
<td>D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td>Op</td>
<td>L1</td>
<td>B1</td>
<td>D1</td>
<td>B2</td>
<td>D2</td>
</tr>
</tbody>
</table>

The values L1, L2, and L are stored by assembler in the machine instruction decremented by 1 (except if the value is 0). In descriptions we refer to the value as written in the assembler instruction, before decrementing.

Effective address of form D1(B1) or D1(X1,B1) is calculated: value of register B1 (take 0 if B1=0 or missing) + value of register X1 (take 0 if X1=0 or missing) + displacement D1 (12-bit constant in range 0..4095).

Condition code (bits 18-19 in PSW) changes are usually as follows:
Arithmetic - 0 if result is zero, 1 if result is negative (highest bit set), 2 if result is positive (highest bit not set), in some cases 3 if there was overflow during operation;
Logical - 0 if result is zero, 1 if result is not zero;
Comparison - 0 if operands equal, 1 if first is strictly lower, 2 if first is strictly greater;

Instructions that work with even/odd register pairs almost always require the specified register to be even. If a 64-bit value is formed from a register pair, the even (lower) register is used as higher 32-bit portion and the odd (higher) register is used as lower 32-bit portion.

Load/Store Instructions

IC  R1,D2(X2,B2)  [43,RX]  
Copy byte at address D2(X2,B2) to lowest byte of register R1. Other bytes of R1 are unchanged. CC: no change.

ICM R1,M3,D2(B2)  [BF,RS]  
Insert consecutive bytes from address D2(B2) into register R1, depending on 4-bit mask M3. For each 1 in mask, a byte is read from memory and copied at the corresponding location in the register. For each 0 in mask, no memory is read and the corresponding byte in register is unchanged. CC: arithmetic by value of the selected bytes as a signed integer.

L  R1,D1(X3,B3)  [58,RX]  
Load full-word from address D1(X3,B3) into register R1. CC: no change.

LA  R1,D2(X3,B2)  [41,RX]  
Load effective address D2(X3,B2) into register R1. The unused high bits of the address (depends on addressing mode) are cleared. Doesn't access memory nor check the address! CC: no change.

LCR R1,R2  [13,RR]  
Load signed value in register R2 into register R1, with opposite sign (change sign). CC: arithmetic. 3 on overflow.

LH  R1,D2(X3,B3)  [48,RX]  
Load signed half-word from address D2(X3,B3) into register R1 (and extend the sign). CC: no change.

LHI R1,I2  [A78,RI]  
Load signed half-word I2 into register R1 (and extend the sign). CC: no change.

LM  R1,R3,D2(B2)  [98,RS]  
Load values into several registers numbered R1 through R3 inclusive from subsequent full-words starting at address D2(B2). CC: no change.

LNR R1,R2  [11,RR]  
Load signed value in register R2 into register R1 with negative sign. CC: arithmetic.

LPR R1,R2  [10,RR]  
Load signed value in register R2 into register R1 with positive sign (absolute value). CC: arithmetic. 3 on overflow.

LR  R1,R2  [18,RR]  
Load value in register R2 into register R1. CC: no change.

LTR R1,R2  [12,RR]  
Load value in register R2 into register R1. CC: arithmetic by the value.

MVC D2(L,B2),D2(B2)  [D2,SS]  
Copy L (up to 256) consecutive bytes of memory from address D2(B2) to address D2(B2). The operation is done byte-wise from lower addresses, and the areas may overlap.
Store values of several registers numbered D.

Store lower half-word in register change.

bytes are stored consecutively in memory. For STCM

Store lowest byte in STC

uses even-odd register pairs R1, R1+1, R2, R2+1 as follows:

- R1 - lowest address of target area
- R1+1 - 24-bit length of target area
- R2 - lowest address of source area
- R2+1 - 24-bit length of source area, highest byte is fill character

The amount of bytes copied is given by lower length, and then the target area is filled with fill character up to the specified length. The registers used are changed like the move were done byte by byte. CC: comparison of lengths, 3 if areas overlap.

MVI D1(B1), I2 [92, SI]
Store byte I2 to address D1(B1). CC: no change.

ST R1,D1(X2,B2) [50, RX]
Store full-word in register R1 to address D1(X2,B2). CC: no change.

STC R1,D2(X2,B2) [42, RX]
Store lowest byte in register R1 to address D2(X2,B2). CC: no change.

STCM R1,M3,D3(B3) [BE, RS]
Store selected bytes in register R1 to address D3(B3) depending on 4-bit mask M3. Selected bytes are stored consecutively in memory. For each 1 in mask, the corresponding byte is read from register and copied into memory. For each 0 in mask, no store is done. CC: no change.

STH R1,D2(X2,B2) [40, RX]
Store lower half-word in register R1 to address D2(X2,B2). CC: no change.

STM R1,R3,D2(B3) [90, RS]
Store values of several registers numbered R1 through R3 inclusive to subsequent full-words starting at address D2(B3). CC: no change.

### Arithmetic Instructions

**A** R1,D3(X2,B2) [5A, RX]
Add signed full-word at address D3(X2,B2) to value in register R1. CC: arithmetic, 3 on overflow.

**AH** R1,D3(X2,B2) [4A, RX]
Add signed half-word at address D3(X2,B2) to value in register R1. CC: arithmetic, 3 on overflow.

**AHI** R1,I2 [A7A, RI]
Add signed half-word constant I2 to value in register R1. To subtract a constant, use negative value. CC: arithmetic, 3 on overflow.

**AL** R1,D3(X2,B2) [5E, RX]
Add unsigned full-word at address D3(X2,B2) to unsigned value in register R1. CC: low bit of CC is set if the result is nonzero, high bit of CC is set if there was overflow (carry).

**ALR** R1,R2 [1E, RR]
Add unsigned value in register R1 to unsigned value in register R2. CC: same as AL instruction.

**AR** R1,R2 [1A, RR]
Add signed value in register R2 to value in register R1. CC: arithmetic, 3 on overflow.

**C** R1,D3(X2,B2) [59, RX]
Compare signed full-word at address D3(X2,B2) to value in register R1. Doesn't change anything. CC: comparison.

**CH** R1,D3(X2,B2) [49, RX]
Compare signed half-word at address D3(X2,B2) to value in register R1. Doesn't change anything. CC: comparison.

**CR** R1,R2 [19, RR]
Compare signed value in register R2 to value in register R1. Doesn't change anything. CC: comparison.

**D** R1,D3(X2,B2) [5D, RX]
Integer division. Dividend is 64-bit value formed from values in even-odd register pair R1, R1+1. Divisor is signed full-word from address D3(X2,B2). The resulting quotient is stored in register R1+1 and the remainder in register R1. May cause fixed-point divide exception if divisor is 0 or quotient is not small enough.

**DR** R1,R2 [1D, RR]
Integer division, works almost same as D instruction, except the divisor is value in register R2. CC: no change.

**M** R1,D2(X2,B2) [5C, RX]
Multiplication. The multiplicand is value in register R1+1 and the multiplier is full-word at address D2(X2,B2). The result is a 64-bit signed value stored into even-odd register pair R1, R1+1. The value in register R1 doesn't affect the result! CC: no change.

**MH** R1,D2(X2,B2) [4C, RX]
Multiply signed value in register R1 by signed half-word at address D2(X2,B2). The result is stored in R1, overflow is ignored. CC: no change.

**MHI** R1,I2 [A7C, RI]
Multiply signed value in register R1 by signed half-word constant I2. The result is stored in R1, overflow is ignored. CC: no change.

**MR** R1,R2 [1C, RR]
Multiplication, works almost same as M instruction, except the multiplier is value in register R2 (can be one of the registers in the even-odd register pair specified by R1). CC: no change.

**S** R1,D2(X2,B2) [5B, RX]
Subtract signed full-word at address D2(X2,B2) from value in register R1. CC: arithmetic, 3 on overflow.

**SH** R1,D2(X2,B2) [4B, RX]
Subtract signed half-word at address D2(X2,B2)
from value in register \( R_1 \). CC: arithmetic, 3 on overflow.

### Logical Instructions

**CL** \( R_1, D_2(X_2, B_3) \) [55,RX]

Compare unsigned full-word at address \( D_2(X_2, B_3) \) to unsigned value in register \( R_1 \).


**CLC** \( D_1(L,B_1), D_2(B_3) \) [D5,SS]

Compare \( L \) (up to 256) consecutive bytes of memory areas at addresses \( D_1(L,B_1) \) and \( D_2(B_3) \). The operation is done byte-wise from lower addresses (lexicographic comparison). CC: unsigned comparison of the first differing byte values, 0 if areas are equal.

**CLC** \( R_1, R_2 \) [0F,RR]

Compare two memory areas. Uses even-odd register pairs \( R_1, R_1+1, R_2, R_2+1 \) as follows:

- \( R_1 \): lowest address of first area
- \( R_2 \): lowest address of second area
- \( R_1+1 \): 24-bit length of first area
- \( R_2+1 \): 24-bit length of second area, highest byte is fill character

The amount of bytes compared is given by higher length (the shorter area is extended with fill character). The registers used are changed like the comparison was done byte by byte, until differing bytes are encountered. CC: unsigned comparison of the first differing bytes, 0 if areas are equal.

**CLM** \( R_1, M_3, D_2(B_3) \) [BD,RS]

Compare selected bytes in register \( R_1 \) to consecutive bytes at address \( D_2(B_3) \) depending on 4-bit mask \( M_3 \). For each 1 in mask, the corresponding byte is compared with a byte in memory (and memory pointer incremented for next byte). For each 0 in mask, no comparison is done. CC: unsigned comparison of the first differing byte selected by mask, 0 if all selected bytes are equal.

**CLR** \( R_1, R_2 \) [15,RR]

Compare unsigned value in register \( R_2 \) to unsigned value in register \( R_1 \). Doesn't change anything. CC: comparison.

**CL** \( R_1, D_2(X_2, B_3) \) [55,RX]

Compare unsigned full-word at address \( D_2(X_2, B_3) \) to unsigned value in register \( R_1 \).


**N** \( R_1, D_2(X_2, B_3) \) [D4,SS]

Logically AND register \( R_1 \) with full-word at address \( D_2(X_2, B_3) \). CC: logical.

**NC** \( D_1(L,B_1), D_2(B_3) \) [D4,SS]

Logically AND \( L \) (up to 256) consecutive bytes of memory at address \( D_1(L,B_1) \) with \( L \) consecutive bytes of memory at address \( D_2(B_3) \). The operation is done byte-wise from lower addresses, and the areas may overlap. CC: 0 if all bytes are zero, 1 otherwise.

**NI** \( D_1(B_3), I_2 \) [94,SI]

Logically AND byte at address \( D_1(B_3) \) with constant byte \( I_2 \). CC: logical.

**NR** \( R_1, R_2 \) [14,RR]

Logically AND register \( R_1 \) with register \( R_2 \). CC: logical.

**O** \( R_1, D_2(X_2, B_3) \) [56,RX]

Logically OR register \( R_1 \) with full-word at address \( D_2(X_2, B_3) \). CC: logical.

**OC** \( D_1(L,B_1), D_2(B_3) \) [D6,SS]

Logically OR \( L \) (up to 256) consecutive bytes of memory at address \( D_1(L,B_1) \) with \( L \) consecutive bytes of memory at address \( D_2(B_3) \). The operation is done byte-wise from lower addresses, and the areas may overlap. CC: 0 if all bytes are zero, 1 otherwise.

**OI** \( D_1(B_3), I_2 \) [96,SI]

Logically OR byte at address \( D_1(B_3) \) with constant byte \( I_2 \). CC: logical.

**OR** \( R_1, R_2 \) [16,RR]

Logically OR register \( R_1 \) with register \( R_2 \). CC: logical.

**SLA** \( R_1, D_2(B_3) \) [8B,RS]

Shift the bits of register \( R_1 \) to the left arithmetically (protect the sign bit). The amount to be shifted is given by low 6 bits of effective address \( D_2(B_3) \). CC: arithmetic.

**SLD** \( R_1, D_2(B_3) \) [8D,RS]

Shift the bits of 64-bit value formed from even-odd register pair \( R_1, R_1+1 \) to the left arithmetically (protect the sign). The amount to be shifted is given by low 6 bits of effective address \( D_2(B_3) \). CC: arithmetic of the 64-bit value.

**SLDL** \( R_1, D_2(B_3) \) [8D,RS]

Shift the bits of 64-bit value formed from even-odd register pair \( R_1, R_1+1 \) to the left. The amount to be shifted is given by low 6 bits of effective address \( D_2(B_3) \). CC: no change.

**SLL** \( R_1, D_2(B_3) \) [89,RS]

Shift the bits of register \( R_1 \) to the left. The amount to be shifted is given by low 6 bits of effective address \( D_2(B_3) \). CC: no change.

**SRA** \( R_1, D_2(B_3) \) [8A,RS]

Shift the bits of register \( R_1 \) to the right arithmetically (extend the sign bit). The amount to be shifted is given by low 6 bits of effective address \( D_2(B_3) \). CC: arithmetic.
SRDA \textit{R}_1, D_2(B_2) \hspace{1cm} [8E,RS]
Shift the bits of signed 64-bit value formed from even-odd register pair \textit{R}_1, \textit{R}_1+1 to the right arithmetically (extend the sign). The amount to be shifted is given by low 6 bits of effective address \textit{D}_2(B_2). CC: arithmetic of the 64-bit value.

SRDL \textit{R}_1, D_2(B_2) \hspace{1cm} [8C,RS]
Shift the bits of 64-bit value formed from even-odd register pair \textit{R}_1, \textit{R}_1+1 to the right. The amount to be shifted is given by low 6 bits of effective address \textit{D}_2(B_2). CC: no change.

SRL \textit{R}_1, D_2(B_2) \hspace{1cm} [88,RS]
Shift the bits of register \textit{R}_1 to the right. The amount to be shifted is given by low 6 bits of effective address \textit{D}_2(B_2). CC: no change.

TM \textit{D}_1(B_1), l_2 \hspace{1cm} [91,SI]
Test the bits of byte at address \textit{D}_1(B_1) according to mask byte \textit{l}_2. Only the bits which have corresponding bit set in the mask are tested. CC: 0 if tested bits are all zero, 1 if tested bits are mixed (some zero, some one), 2 if tested bits are all one.

\textbf{Branch Instructions}

\textbf{BAL} \textit{R}_1, D_3(X_2, B_2) \hspace{1cm} [45,RX]
Store the address of the next instruction (from PSW) into register \textit{R}_1 and then branch to address \textit{D}_3(X_2, B_2). Depending on addressing mode, unused high bits of address stored into \textit{R}_1 will contain other information from PSW. CC: no change.

\textbf{BALR} \textit{R}_1, R_2 \hspace{1cm} [05,RR]
Similar to the BAL instruction, except the address of branch is taken from register \textit{R}_2 (but when \textit{R}_2 is zero, no branch is taken).

\textbf{BAS} \textit{R}_1, D_3(X_2, B_2) \hspace{1cm} [4D,RX]
Store the address of the next instruction (from PSW) into register \textit{R}_1 and then branch to address \textit{D}_3(X_2, B_2). Unlike in BAL instruction, unused high bits of the address (depending on addressing mode) are cleared. CC: no change.

\textbf{BASR} \textit{R}_1, R_2 \hspace{1cm} [0D,RR]
Similar to the BAS instruction, except the address of branch is taken from register \textit{R}_2 (but when \textit{R}_2 is zero, no branch is taken).

\textbf{BC} \textit{M}_1, D_3(X_2, B_2) \hspace{1cm} [47,RX]
Branch to address \textit{D}_3(X_2, B_2) depending on current condition code and 4-bit mask \textit{M}_1. Each bit of mask corresponds to possible value of CC. The branch is taken if the bit of the mask corresponding to the current value of condition code is set. Assembler provides additional mnemonics for various common mask types. CC: no change.

\textbf{BCR} \textit{M}_1, R_2 \hspace{1cm} [07,RR]
Similar to the BC instruction, except the address of branch is taken from register \textit{R}_2 (but when \textit{R}_2 is zero, no branch is taken).

\textbf{BCT} \textit{R}_1, D_3(X_2, B_2) \hspace{1cm} [46,RX]
Decrement value in register \textit{R}_1. If the result is not zero, branch to address \textit{D}_3(X_2, B_2). CC: no change.

\textbf{Branch Instructions}

\textbf{BCTR} \textit{R}_1, R_2 \hspace{1cm} [06,RR]
Similar to the BCT instruction, except the address of branch is taken from register \textit{R}_2 (but when \textit{R}_2 is zero, no branch is taken).

\textbf{BXH} \textit{R}_1, R_3, D_2(B_2) \hspace{1cm} [86,RS]
Add value from register \textit{R}_3 (should denote even-odd register pair) to register \textit{R}_1. The result is then compared to register \textit{R}_3+1 and if it is higher, branch is taken to address \textit{D}_2(B_2). Usually used to loop over a table backwards, so registers are set as follows: \textit{R}_1 - address of last/current entry \textit{R}_3 - negative length of table entry \textit{R}_3+1 - address of table minus one CC: no change.

\textbf{BXLE} \textit{R}_1, R_3, D_2(B_2) \hspace{1cm} [87,RS]
Add value from register \textit{R}_3 (should denote even-odd register pair) to register \textit{R}_1. The result is then compared to register \textit{R}_3+1 and if it is less or equal, branch is taken to address \textit{D}_2(B_2). Usually used to loop over a table, so registers are set as follows: \textit{R}_1 - address of first/current entry \textit{R}_3 - length of table entry \textit{R}_3+1 - address of last entry in table CC: no change.

\textbf{Decimal Instructions}

\textbf{Packed decimal}

\textbf{Packed decimal} numbers have 1-31 decimal digits. Each digit is stored in 4 bits, and last 4 bits of packed number is a sign - either C (positive), D (negative) or F (positive). Packed decimal of length \(k\) bytes has \(2^k-1\) digits. The length of number (in bytes) is included with the instruction. If the packed decimal has incorrect sign bits, it is invalid!

\textbf{AP} \textit{D}_1(L_1, B_1), D_2(L_2, B_2) \hspace{1cm} [FA,SS]
Add packed decimal number at address \textit{D}_1(B_1) (of length \textit{L}_2 bytes) to packed decimal number
at address \( D_1(B_1) \) (of length \( L_1 \) bytes). CC: arithmetic, 3 on overflow.

**CP** \( D_1(L_1,B_1),D_2(L_2,B_2) \) [F9,SS]

Compare packed decimal number at address \( D_1(B_1) \) (of length \( L_1 \) bytes) to packed decimal number at address \( D_1(B_1) \) (of length \( L_1 \) bytes). CC: comparison.

**CVB** \( R_2(D_2)(X_2,B_2) \) [4F,RX]

Convert packed decimal number at address \( D_1(B_1) \) (of length 8 bytes) to signed binary integer into register \( R_2 \). CC: no change.

**CVD** \( R_2(D_2)(X_2,B_2) \) [4E,RX]

Convert signed binary integer in register \( R_2 \) to packed decimal number at address \( D_1(B_1) \) (of length 8 bytes). CC: no change.

**DP** \( D_1(L_1,B_1),D_2(L_2,B_2) \) [FD,SS]

Decimal integer division. Dividend is packed decimal number at address \( D_1(B_1) \) (of length \( L_1 \) bytes) and divisor is packed decimal number at address \( D_2(B_2) \) (of length \( L_2 \) bytes). The result is quotient is stored as a packed decimal from address \( D_1(B_1) \) and has length \( L_1\cdot L_2 \), and the resulting remainder is stored next as a packed decimal of length \( L_2 \) (so the result is in place of original dividend). The following conditions must hold: \( L_1\leq16 \), \( L_2\leq9 \), \( L_1,L_2\leq L_2 \). CC: no change.

**ED** \( D_1(L_1,B_1),D_2(L_2,B_2) \) [DE,SS]

Format packed decimal number at address \( D_2(B_2) \) according to mask at address \( D_2(B_2) \) (of length \( L_1 \) bytes). The mask is modified byte-wise from left, and the packed number is read digit-wise from left. First byte of the mask is fill byte (unchanged). During execution, there is binary internal state SS - “significance start”, 0 at the start. The interpretation of mask bytes is as follows:

X’20’ - digit select - a digit from the packed decimal number is consumed; if the digit is zero and SS=0, fill character is printed into mask; otherwise the digit character is printed; if the last digit is consumed, SS is set depending on sign half-byte (0 for +, 1 for -)

X’21’ - digit select with significance start - like digit select, but set SS=1 for the next digit (unless this was last)

X’22’ - field separator

other - skipped if SS=1, replaced by fill character if SS=0

CC: roughly arithmetic by number edited.

**EDMK** \( D_1(L_1,B_1),D_2(L_2,B_2) \) [DF,SS]

Similar to the ED instruction, but will store address of first significant digit into register 1. If SS stays 0 or was forced by X’21’, register 1 is unchanged. CC: like ED instruction.

**MP** \( D_1(L_1,B_1),D_2(L_2,B_2) \) [FC,SS]

Multiply packed decimal number at address \( D_1(B_1) \) (of length \( L_1 \) bytes) by packed decimal number at address \( D_2(B_2) \) (of length \( L_2 \) bytes). The following conditions must hold: \( L_1\leq8 \), \( L_2\leq1 \). The first operand must have at least \( L_2 \) leftmost zero bytes. CC: no change.

**MVN** \( D_1(L_1,B_1),D_2(L_2,B_2) \) [D1,SS]

Copy low half-bytes (numeric fields) of \( L \) (up to 256) consecutive bytes of memory from address \( D_1(B_1) \) to address \( D_2(B_2) \). The operation is done byte-wise from lower addresses, and the areas may overlap. High half-bytes in target area are not changed. CC: no change.

**SP** \( D_1(L_1,B_1),D_2(L_2,B_2) \) [FB,SS]

Subtract packed decimal number at address \( D_2(B_2) \) (of length \( L_2 \) bytes) from packed decimal number at address \( D_1(B_1) \) (of length \( L_1 \) bytes). CC: arithmetic, 3 on overflow.

**SRP** \( D_1(L_1,B_1),D_2(B_2),I_3 \) [F0,SS]

Shift digits of (and round) packed number at address \( D_1(B_1) \) (of length \( L_1 \) bytes). The amount to shift to the left is given by low 6 bits of effective address \( D_2(B_2) \), interpreted as signed number (so negative values shift to right). Shift to left will insert 0 digits from right. After shift to right, low 4 bits of \( I_3 \) are added to the leftmost lost digit, which accomplishes rounding of the result:

- \( I_3=0 \) to round down
- \( I_3=5 \) to round normally
- \( I_3=9 \) to round up

CC: arithmetic, 3 on overflow.
UNPK \( D_1(L_1,B_1), D_2(L_2,B_2) \) \([F3,SS]\)

Convert packed decimal number at address \( D_1(B_2) \) (of length \( L_2 \) bytes) to zoned decimal number at address \( D_1(B_1) \) (of length \( L_1 \) bytes). Packed number is processed digit-wise from right. The rightmost 2 half-bytes of packed number are stored swapped into the rightmost byte of zoned number. Then, from each digit of packed number, byte \( x'F0' \) is put as a digit into the zoned number (going from right), until either of them is exhausted. The remainder of the zoned number is filled with \( x'F0' \). There is no check if the numbers are in correct format, it just rearranges the half-bytes. CC: no change.

**Special Instructions**

**CDS** \( R_1, R_3, D_2(B_2) \) \([BB,RS]\)

Similar to the CS instruction, but with double-words. The \( R_1 \) and \( R_3 \) both denote an even-odd register pair, and address \( D_1(B_2) \) is a double-word in memory.

**CS** \( R_1, R_3, D_2(B_2) \) \([BA,RS]\)

Compare the value in \( R_1 \) to full-word at address \( D_2(B_2) \). If equal, store full-word in register \( R_3 \) to address \( D_2(B_2) \) and set CC=0. If not equal, load full-word from address \( D_2(B_2) \) into register \( R_1 \) and set CC=1. The instruction is atomic and can be used for processor synchronization.

**EX** \( R_1, D_2(X_2,B_2) \) \([44,RX]\)

Fetch an instruction from address \( D_2(X_2,B_2) \) (must be even) and then logically OR the 2nd byte (usually length specifier) of the instruction with the lowest byte of register \( R_1 \) and execute the resulting instruction. The execution then continues normally with the next instruction after EX (unless the executed instruction altered PSW). Doesn't alter the instruction in memory in any way. CC: depends on target instruction.

**STCK** \( D_1(B_1) \) \([B205,S]\)

Store current value of machine TOD clock to address \( D_1(B_1) \). The TOD clock value has 64 bits, and represents time since 00:00:00 AM of 1st January 1900. Bit 51 has resolution 1 microsecond, and bit 31 (lowest bit of the first full-word) has resolution 1.048576 seconds. CC: 0 if value stored was valid.

**SVC** \( I \) \([0A,I]\)

Supervisor call - invoke an operating system service number \( I \) (0-255). The current PSW is saved, and another PSW is retrieved instead from system location. Anything can be changed upon return, depends on the service routine. In z/OS, SVCs 0-127 are reserved for the system, while SVCs128-255 can be defined by installation or other programs. Usually, registers 0,1,14,15 are used to pass parameters to and from SVC routines.

**TS** \( D_1(B_1) \) \([93,S]\)

Set CC with the value of leftmost bit of byte at address \( D_1(B_1) \), then set the byte with \( x'FF' \). The instruction is atomic and can be used to obtain a lock.

### Branch Mnemonics

Assembler provides additional mnemonic for BC instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Equivalent</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>B addr</td>
<td>BC 15,addr</td>
<td>Always</td>
</tr>
<tr>
<td>NOP addr</td>
<td>BC 0,addr</td>
<td>Never</td>
</tr>
<tr>
<td>BE addr</td>
<td>BC 8,addr</td>
<td>Op1 = Op2</td>
</tr>
<tr>
<td>BNE addr</td>
<td>BC 7,addr</td>
<td>Op1 != Op2</td>
</tr>
<tr>
<td>BL addr</td>
<td>BC 4,addr</td>
<td>Op1 &lt; Op2</td>
</tr>
<tr>
<td>BNL addr</td>
<td>BC 11,addr</td>
<td>Op1 &gt;= Op2</td>
</tr>
<tr>
<td>BH addr</td>
<td>BC 2,addr</td>
<td>Op1 &gt; Op2</td>
</tr>
<tr>
<td>BHL addr</td>
<td>BC 13,addr</td>
<td>Op1 &lt;= Op2</td>
</tr>
<tr>
<td>BZ addr</td>
<td>BC 8,addr</td>
<td>Result = 0 (or all bits zeros in TM)</td>
</tr>
<tr>
<td>BNZ addr</td>
<td>BC 7,addr</td>
<td>Result != 0 (or not all bits zeros in TM)</td>
</tr>
<tr>
<td>BM addr</td>
<td>BC 4,addr</td>
<td>Result &lt; 0 (or bits mixed in TM)</td>
</tr>
<tr>
<td>BNM addr</td>
<td>BC 11,addr</td>
<td>Result &gt;= 0 (or bits not mixed in TM)</td>
</tr>
</tbody>
</table>
Similar mnemonics exists for BCR instruction (just add "R").

Assembler Data Types

<table>
<thead>
<tr>
<th>Spec</th>
<th>Type</th>
<th>Length</th>
<th>Align</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(...)</td>
<td>Address expression</td>
<td>4 (1-4)</td>
<td>4</td>
<td>Absolute or relocatable</td>
</tr>
<tr>
<td>B'...</td>
<td>Binary integer</td>
<td>any (1-256)</td>
<td>1</td>
<td>Binary, pads 0s left</td>
</tr>
<tr>
<td>C'...</td>
<td>Character string</td>
<td>any (1-256)</td>
<td>1</td>
<td>Pads blanks right</td>
</tr>
<tr>
<td>D'...</td>
<td>Long float</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>F'...</td>
<td>Full-word</td>
<td>4</td>
<td>4</td>
<td>Decimal int</td>
</tr>
<tr>
<td>FD'...</td>
<td>Double-word</td>
<td>8</td>
<td>8</td>
<td>Decimal int</td>
</tr>
<tr>
<td>H'...</td>
<td>Half-word</td>
<td>2</td>
<td>2</td>
<td>Decimal int</td>
</tr>
<tr>
<td>P'...</td>
<td>Packed decimal</td>
<td>any (1-16)</td>
<td>1</td>
<td>Decimal, pads 0s left</td>
</tr>
<tr>
<td>S(...)</td>
<td>Base reg. + displacem.</td>
<td>2</td>
<td>2</td>
<td>Address operand</td>
</tr>
<tr>
<td>V(...)</td>
<td>External address</td>
<td>4 (1-4)</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>X'...</td>
<td>Hex integer</td>
<td>any (1-256)</td>
<td>1</td>
<td>Hex, pads 0s left</td>
</tr>
<tr>
<td>Y(...)</td>
<td>Address expression</td>
<td>2 (1-2)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Z'...</td>
<td>Zoned decimal</td>
<td>any (1-16)</td>
<td>1</td>
<td>Decimal, Pads 0s left</td>
</tr>
</tbody>
</table>

Assembler Instructions

**label AMODE [24|31|64|ANY]**
Set addressing mode attribute of section starting at label.

**label CSECT**
Start a new control section named label.

**label DC [r]type[Llen][value]**
Define data constant. Type of constant is given by type type (see table). Default length can be changed to len. Converts value to appropriate representation and stores it within the program (advancing location counter). The expression will be repeated r times.

**label DS [r]type[Llen][value]**
Define space for data. Type of constant is given by type type (see table). Default length can be changed to len. Discards the actual value, only advances location counter by its length. The expression will be repeated r times.

**label DSECT**
Start a new dummy section named label.

**DROP [reg][label]...**
Undefine previous USING instruction. Operand is either register reg currently used as a base or label of previous USING (if it was labeled, then you must use the label). More than one registers/labels to drop can be specified at once.

**EJECT**
Start printing a new page in the listing.

**END**
End of assembly.

**label EQU addr[,len]**
Associate an address expression (absolute or relocatable) addr, and optionally length len, with symbol label.

**LTORG**
Force generation of literal pool. Normally, literal pool is generated at the end of section.

**ORG [addr]**
Change the current location counter to point to relocatable address expression addr. If addr is omitted, change the value of location counter behind the last generated location.

**POP [PRINT|USING][,NOREPRINT]**
Restore the PRINT or USING status from the stack.

**PRINT ops,...[,NOREPRINT]**
Select which statements will be printed into the listing, where ops is comma delimited list of operands, can be: ON, OFF, [NO]GEN, [NO]DATA, [NO]MCALL.

**PUSH [PRINT|USING][,NOREPRINT]**
Save the PRINT or USING status to stack.

**RMODE [24|31|ANY]**
Set residence mode attribute of section starting at label.

**SPACE**
Print a space in the listing.

**TITLE 'string'**
Change the page header in listing to string.

**label USING base,reg[,...]**
Define to assembler that register reg will contain address base (which may be a name of a section) and can be used as base register when resolving symbols to addresses. More than one register can be specified, then they are assumed to be incremented by 4K with respect to the previous one. If label is specified, then same symbol in two overlapping USING ranges can be resolved using the label.symbol notation.

**label USING base,addr**
Define dependent using. The symbols starting from the address base (which may be a name of a section) will be resolved to addresses by means that addr is resolved.
**z/OS Call Conventions**

**Standard save area (18 words):**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>Unused</td>
</tr>
<tr>
<td>+4</td>
<td>Previous SA (up)</td>
</tr>
<tr>
<td>+8</td>
<td>Next SA (down)</td>
</tr>
<tr>
<td>+12</td>
<td>Return address (R14)</td>
</tr>
<tr>
<td>+16</td>
<td>Entry point (R15)</td>
</tr>
<tr>
<td>+20</td>
<td>Register R0</td>
</tr>
<tr>
<td>+24</td>
<td>Register R1</td>
</tr>
<tr>
<td>...</td>
<td>other registers</td>
</tr>
<tr>
<td>+68</td>
<td>Register R12</td>
</tr>
</tbody>
</table>

**Typical register usage:**
- R0: additional value on exit
- R1: address of parameter list on entry
- R13: address of save area provided for subroutine
- R14: return address
- R15: entry point address on entry, return code on exit

**Typical entry code:**
```assembly
CSECT
STM R14,R12,12(R13)
BASR R12,0
USING *
LA R2,SAVEAREA
ST R2,8(R13)
ST R13,4(R2)
LR R13,R2
...
SAVEAREA DS 18F
```

**Typical exit code:**
```assembly
...
L R13,4(R13)
L R14,12(R13)
LM R0,R12,20(R13)
BR R14
```

**Typical call code:**
```assembly
LA R1,PLIST
L R15,ASUB
BASR R14,R15
...
ASUB DC V(SUB)
PLIST DC A(PARM1)
DC A(PARM2)
...
DC A(PARMN+X'80000000')
```

**Common z/OS Macros**

**Note:** Many parameters and options are omitted, only the most common are listed. Usually, where macros expect an address, you can also use register 2-12 in parentheses (macros use LA to get the address).

**ABEND compcode,[DUMP],[STEP]**
Abend current task (or job step if STEP is specified) with user ABEND code `compcode` (value 0-4095), and take a dump if DUMP is specified.

**CALL entry,(addr,...),[VL]**
Call a subroutine at `entry` (symbol or register (15)). The `addr` is one or more addresses (symbols) that will be referenced from the parameter list created before call. If VL is coded, then last address of parameter list will have highest bit set (end of list).

**CLOSE (dcb)**
Close open DCB at address `dcb`.

**DCB DDNAME=ddname,**
```assembly
MACRF=[P|G][M|L],
DSORG=PS,RECFM=recfm,
LRECL=lrecl,BLKSIZE=blksize,
EODAD=eodad
```
Generate DCB (data control block) for a sequential dataset (QSAM). The `ddname` is a string used as DD name. The MACRF denotes macros being used - either GET or PUT in move or locate mode. Parameters `recfm`, `lrecl` and `blksize` are same as in JCL. The `eodad` parameter is address where the branch is made if the EOF is encountered during GET.

**GET dcb[,area]**
Read an input record from the open DCB at address `dcb`. In move mode, copy the record to address `area`. In locate mode, return the address of the record read in register 1.

**OPEN (dcb,[(INPUT|OUTPUT)])**
Open DCB at address `dcb` for input or output, as specified. The unspecified parameters in DCB are filled first from the JCL and then from dataset attributes.

**PUT dcb[,area]**
Write a record into the open DCB at address `dcb`. In move mode, write an output record from address `area`. In locate mode, return in register 1 address where the next output record should be written.

**RETURN (reg1,reg2),[T],RC=rc**
Return from a subroutine. Restore registers `reg1` through `reg2` from save area (in R13) and branch to address in register R14 (return). If T is coded, the save area is marked as used. If RC is coded, return code `rc` is stored in R15 upon return.

**SAVE (reg1,reg2),[T],'identifier'**
Save registers `reg1` through `reg2` into the save area (in R13) upon entry to the subroutine. If T is coded, R14 and R15 will also be saved. The subroutine will have identifier string in its header.

**STORAGE OBTAIN,LENGTH=len**,`[ADDR=addr][,LOC=24]`
Obtain block of memory of length `len` (symbol or register (0),(2)-(12)) from the system. Returns address of memory to address `addr` (symbol or register (1)-(12), default register 1). If LOC=24 is coded, the memory is
allocated under line, otherwise it is allocated depending on program residency.

**STORAGE RELEASE,LENGTH=len, ADDR=addr**

Release storage at address *addr* (symbol or register (1)-(12)) of length *len* (symbol or register (0),(2)-(12)) back to the system.

**YREGS**

Define register equates (so one can write R0 instead of 0 and so on).

**Common z/OS ABEND Codes**

**S0C1** Program Check - Operation Exception:
The processor's PSW is addressing a byte of storage that is NOT a valid "opcode". Likely reasons: (1) you have placed data in the path of execution, (2) you have "branched" to a data area instead of an executable instruction, (3) you have overlaid valid instructions with data.

**S0C4** Program Check - Addressing Exception:
The instruction is attempting to access data that is outside the range of valid addresses for your execution. Likely reason: (1) the content of a register being used as a base to a field of data has become invalid, or (perhaps) was never valid.

**S0C6** Program Check - Specification Exception:
The processor's PSW is pointing at an instruction that has some fundamental problem. Likely reasons: (1) you have branched to an odd address (instructions must be on half-word boundaries), (2) you have used an odd register where an even-numbered register is expected.

**S0C9** Program Check - Fixed-point Divide Exception:
A 'D' or 'DR' instruction was attempted, but could not be completed. Likely reasons: (1) The divisor has value zero, (2) the dividend is too large in its even/odd register pair such that the quotient will not fit into a 32-bit register.

**S222** Task canceled by operator:
Operator issued a system command to cancel the task.

**S322** Task terminated by system (TIME parameter exceeded):
The JOB statement had a TIME parameter, and your execution exceeded the specified limit. Likely reason: (1) the program is in a "loop;", or (2) the program is legitimately a long-running task and the default time threshold was exceeded.

The current instruction is expecting valid packed decimal data, and has found a field that doesn't conform to packed decimal rules for digits and sign. Likely reasons: (1) the data field was not packed, or (2) the data has been packed, but the original data was not digits, or (3) a packed decimal field has been corrupted, or (4) the reference to a packed decimal field has the wrong length.