Midterm

Nathan Warner

Northern Illinois University

Computer Science
Northern Illinois University
United States

Contents

Facts

- **Positive and negative hex numbers**: Leftmost hex digit 0-7: positive. 8-F: negative
- **Instructions and their lengths**: Look at the leftmost hex digit
  - **0,1,2,3**: 2 byte
  - **4-B**: 4 byte
  - **C-F**: 6 byte
- **Exceptions**
  - **S0C 1 (Operation)**: Invalid opcode / instruction
  - **SOC 4 (Protection)**: Accessed memory outside scope of program
  - **SOC 5 (Addressing)**: Invalid address
  - **SOC 6 (Specification)**: Operand not on FWB
- **Which instructions require FWB operands (might cause S0C6?)**: All RX
- **The byte in an SI instruction**: Given in one of four ways, a character, a two digit hex number, an eight bit binary number, or a decimal between 0 and 255, which will be converted to hex in the encoding

```
MVI   42(15),C'$'
MVI   42(15),X'5B'
MVI   42(15),B'01110110'
MVI   42(15),32
```

- **EQU**: EQU, or EQUates, assigns a value to a label

```
label    EQU    Expression
```

gives a label the value of the expression. Every occurrence of label will be treated as if it was the expression.

Equates are either typed above the CSECT or below the END

```
LOAD  EQU  L
```

Then, a load instruction can be written as

```
LOAD   3,NUM1
```

Before assembling the code, the Assembler replaces the label of the equates with the expression.

- **Carriage control**
  - **C' '**: Single space
  - **C'0'**: Double space
  - **C'-'**: Triple space
  - **C'1'**: Top of next page
- **Label that does not occupy storage**

```
label    DS    0H
```

- **Add one to register** *R*

```
LA   R,1(,R)
```

- **Decrement register** *R*

```
BCTR R,0
```

- **Zero out register**:

```
SR   R,R
```

- **FWB**: Rightmost hex digit 0,4,8,C
- **HWB**: Rightmost hex digit 0,2,4,6,8,A,C,E
- **DWB**: Rightmost hex digit 0,8
- **LTORG starts on DWB**
- **Max displacement**: FFF, or 4095
- **Read loop**:

```
        XREAD RECORD,80
LOOP1   BC    B'0111', ENDLOOP1
        XPRNT DETAIL,133
        XREAD RECORD,80
        BC    B'1111',LOOP1
        ENDLOOP1
```

- **Default sign digits**:
  - **Positive:** C
  - **Negative:** D

- **Determine how many bytes of packed decimal storage you will need to declare to hold a converted zoned decimal number**: We take

$$\lfloor \text{len(zoned decimal bytes)}/2 \rfloor + 1.$$

- **SRP syntax**:

  Left shift

  ```
  SRP PNUM2(11),4,0
  ```

  Right shift:

  ```
  SRP PNUM3(10),64-3,5
  ```

- **Decker's Rules for Packed Decimal Instructions**:
  - Begin the label, or name, of ALL packed decimal fields with the letter P, for Packed
  - Declare ALL packed decimal fields with a DC, a specific length in bytes and initialized to 0 (unless to some other value)
  - NEVER let lengths default! Code a length on every packed operand where it CAN be coded!
- **Errors with packed decimals**:
  - **Data Exception (S0C7)**: At least one of the operands is not a valid packed decimal representation
  - **Decimal-overflow Exception (S0CA)**: The result is too large for the receiving field
- **Errors that occur with MP**:
  - **Specification Exception (S0C6)**: – if length of second operand is greater than 8 or if length of the second operand, the multiplier, is greater than length of the first operand, the multiplicand
  - **Data Exception – (S0C7)**: if the first n bytes of the first operand are not all zeros where n is the length of the second operand or if at least one of the operands is not a valid packed decimal number
- **Errors that occur with DP**:
  - **Specification Exception (S0C6)**: if length of second operand is greater than 8 or if length of the second operand is greater than or equal to the length of the first operand.
  - **Decimal-divide Exception (S0CB)**: if quotient will not fit in n bytes where n is the length of the first operand minus the length of the second operand.
  - **Data Exception (S0C7)**: if at least one of the operands is not a valid packed decimal number
- **Errors that occur with SRP**:

  - **Decimal-overflow Exception (S0CA)**: if a left shift results in losing nonzero digits.

- **Errors that occur with CVB**:
  - **Specification Exception – S0C6 –**: if the second operand is not on a doubleword boundary.
  - **Data Exception – S0C7 –**: if the doubleword does not hold a valid packed decimal number.
  - **Fixed-point Divide Exception – S0C9** – if the packed decimal number at the D(X,B) address is too large to be represented in 32 SIGNED bits in the register.

- **Errors that occur with CVD**:
  - **Specification Exception – S0C6 –**: if the second operand is not on a doubleword boundary.

- **Notes about ED and EDMK**: Digits are moved from the source field, one at a time, and from left to right.

  If not enough digit selectors, the result will be truncated on the right

  A Data Exception – S0C7 – can occur if the source field is not a valid packed decimal number.

- **RS instructions**: RS instructions are registers to storage, they have the form

```
0  NAME   R1,R2,D(B)
```

- **Internal vs external subroutines**: An internal subroutine in Assembler is one that is located within a single control section, or CSECT.

  External subroutines

  - It is located outside of the caller's CSECT.
  - It has its own CSECT.
  - Is actually a subprogram (although it's often referred to as a subroutine).
  - External subprograms can be written in C++, Java, etc., on the mainframe

- **Notes about internal subroutines**: There are specific standards that Assembler programmers worldwide must follow to call subroutines and subprograms.

- **Internal subroutine standards**:
  - On entrance to a subroutine, register 1 holds the address of a parameter list (if parameters need to be passed in)
  - On entrance to a subroutine, the initial value in any register that will be altered by the subroutine should be saved using ST and/or STM as necessary
  - Before exiting a subroutine, the initial value in any register that was altered by the subroutine should be restored using L and/or LM as necessary

- **Defining an Internal Subroutine**: The following puts a label in storage without moving the location counter or actually declaring storage:

```
rtnName DS 0H
```

Note that the name of the subroutine can also be placed on the subroutine's first instruction.

```
rtnName STM 3,7,SUBSAVE
```

- **Parameter list**: Standards dictate that a parameter list must be declared in a very specific manner in Assembler.

  Parameters are only passed by reference in Assembler and NEVER by value.

  A parameter list is a set of contiguous fullwords, each containing the address of a parameter, or variable, to be passed.

  To declare a parameter list that allows the addresses in the fullwords in the parameter list to be virtual, we use Address Constants, or ADCONs, defined as

  ```
  label DC A(expression)
  ```

  If **expression** is a non-negative integer, the generated fullword will contain the binary representation of that integer, which is the same as declaring

  ```
  label DC F'expression'
  ```

  If expression is a *label* or $label + n$, the generated fullword will contain the address of label or $label + n$.

  ```
  PARM DC A(5)
  ```

  declares a fullword at label PARM in storage as:

  $$00000005$$

  ```
  PARMLIST DC A(FIELD1)
  ```

  declares a fullword at label PARMLIST in storage as: 00000148, if FIELD1 is declared in storage and is at location counter value 000148 after assembly

  ```
  000148 FIELD1 DC F'34220'
  ```

```
PARMLIST DC A(45,FIELD2)
```

declares two fullwords at label PARMLIST in storage as:

0000002D00000274

if the following is declared in storage and is at location counter value 000274 after assembly

```
000274 FIELD2 DC F'2301.00'
```

Another example declaration:

```
PARMLIST DC A(FIELD3)
         DC A(34)
```

Standards dictate that, if the internal subroutine needs parameters passed into it, set up the parameter list similarly to what is shown immediately above

Then load the address of PARM or PARMLIST into register 1 before calling the subroutine.

By the way, it's the same for passing parameters to external subprograms!

- **Passing control to subroutine**: Here is the sequence of instructions calling an internal subroutine using a parameter lis

```
LA 1,PARMLIST POINT R1 AT PARMLIST
BAL 11,SUBRTN BRANCH AND LINK TO SUBRTN
```

Then, when the subroutine is finished, branch back to the instruction in the caller immediately following the call, or BAL:

```
BR 11 RETURN TO CALLER
```

- **Using parameters**: In the subroutine, after we save the caller's registers, we dereference the parms

```
STM   2,4,SAVEREGS   STORE REGS TO BE USED
LM    2,4,0(1)
```

- **Returning control to caller**: When the subroutine completes its task and is ready to return to the caller, it must:

  - Restore the caller's registers using either a Load (L) if only a single register needs to be restored or Load Multiple (LM) if more than one
  - Return to the caller using: BR 11

- **External subprograms**: Subprograms are similar to internal subroutines but they are outside, or external, to our program.

  In Assembler, they have their own CSECT

  An external subprogram is a type of subroutine but

  - It is located outside of the caller's CSECT.
  - It has its own CSECT.
  - Is actually a subprogram (although it's often mistakenly referred to as a subroutine).
  - External subprograms can be written in COBOL, C, C++, Metal C, Java, etc., on the mainframe

- **Calling a external subroutine**: Here is the sequence of instructions calling an external subprogram using a parameter list:

```
LA   1,PARMLIST    POINT R1 AT PARMLIST
L    15,=V(SUBPGM) LOAD 15 WITH ADDR OF SUBPGM
BALR 14,15         BRANCH AND LINK TO SUBRTN
```

  Then, when the subprogram is finished, branch back to the instruction in the caller immediately following the call, or BALR:

```
BR 14 RETURN TO CALLER
```

- **DROP Statement**:

```
DROP R
```

  Or

```
DROP R1,R2,...,RN
```

  It ends the "domain" of a USING statement. The DROP informs the Assembler that register R or registers R1,R2,...,Rn are no longer to be associated with label.

  Or, that the specified register is no longer (for instructions below) supposed to be used to convert implicit addresses to explicit addresses for encoding instructions.

- **Dummy SECTions, or DSECTs**: A dummy section is used to specify a format that can be associated with a particular area in storage without producing any object code.

The end of a dummy section is signaled by the occurrence of a CSECT statement, another DSECT statement or an END statement

An example DSECT definition:

```
$TABELEM DSECT
$STCKNUM DS    F
$ARTIST  DS    CL24
$TITLE   DS    CL24
$INSTOCK DS    F
$PRICE   DS    F
```

specifies the format of a table element. The labels $STCKNUM, $ARTIST, etc., can be used rather than displacements into the element itself.

Note the convention to use the $ to denote the name of a DSECT and its fields

Before a DSECT can be used, a USING statement must be coded

```
USING $TABELEM,3
```

- **Standard Entry and Exit Linkage Conventions**:
  - Conventions about how to call a subprogram and return from it were standardized many years ago by a group of Assembler developers. What follows is a description of those conventions.
  - Standards dictate that, when control is passed to an external subprogram, register 15 contains the address of the subprogram.
  - Standards dictate that register 14 contains the address of the next instruction to execute in the caller program, i.e., the one following the call to the subprogram, i.e., the instruction.
  - Standards dictate that register 13 contains the address of an 18-fullword save area in the caller's storage in which its own registers will be saved by a called subprogram(!).
  - Just as was presented in the previous chapter when calling internal subroutines, parameters are passed to an external subprogram in exactly the same manner. Standards dictate that register 1 contains the address of the beginning fullword of the parameter list if parameters are passed.
  - Return codes are passed back to the caller in register 15.
  - A simple calculated value can be passed back to the caller in register 0 (rare, and should be avoided).
  - As hinted above while talking about register 13, the subprogram is responsible for storing the contents of the caller's registers upon entry to the routine and restoring those values before returning control to the caller.
- **Format of 18F save area**

Unused
Backward Pointer
Forward pointer
R14
R15
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12

Notice that we do not include R13

- **Standard entry linkage**: The following code should be included as the first lines of each CSECT

```
pgmName    CSECT
           STM   14,12,12(13)
           LR    12,15
           USING pgmName,12
           LA    14,name_of_18F_save_area_in_pgmName
           ST    13,4(,14)
           ST    14,8(,13)
           LR    13,14
```

```
STM   14,12,12(13)
```

Saves all of the caller's registers, except for register 13, in the caller's 18-fullword register save area.

```
LR    12,15
Using pgmName,12
```

Puts the address of pgmName in R12, then establishes R12 as the base register for pgmName

```
LA    14,name_of_18F_save_area_in_pgmName
```

points register 14 to an 18-fullword register save area in the current program, pgmName's, storage where its own registers will be saved if it calls a subprogram itself.

```
ST    13,4(,14)
```

stores the address of the caller's 18-fullword register save area in the current program, pgmName's, own 18- fullword register save area. This value in register 13 is known as the **backward pointer.**

```
ST    14,8(,13)
```

stores the address of the current program, pgmName's, 18-fullword register save area in the caller's 18- fullword register save area. This value in register 14 is known as the **forward pointer**.

```
LR    13,14
```

now points register 13 to the current program, pgmName's, 18-fullword register save area in case it calls a subprogram itself.

- **Standard Exit Linkage**: The following code should be included as the last lines of executable code in each CSECT if no return code is being passed back in register 15 (and no calculated value is being returned in register 0:

```
L    13,4(,13)
LM   14,12,12(13)
BR   14
```

```
L    13,4(,13)
```

- **Tables**: tables in assembler are like arrays in higher-level languages. Like arrays, Assembler tables are defined in storage – and given a name – to store related data items.

  Examples of ”related” data items or data items of ”similar character”

  - the ages of each each individual student in class
  - the account balances of checking accounts of bank customers
  - the daily average temperature in New York City for the calendar year
  - the names of the students in an Assembler class

  In an array, we call the storage for an individual data item an element. In Assembler, this storage is called an entry. In arrays, we can only store one data item per element and, unless an array of objects, all data items must be of the same primitive data type. In Assembler tables, we have complete freedom to store any combination of data items and types in a single entry of a table.

In Assembler, we can define a table in storage using any storage class. We only need to be sure that the table is big enough in bytes to hold all of the data we need it to hold.

Here is an example of an Assembler table definition that can hold integer test scores for up to 50 students

```
SCORES DC 50F'0'
```

Instructions

2.1 RR

- **AR (1A)**: Sets condition code
- **SR (1B)**: Sets condition code
- **LR (18)**:
- **CR (19)**: Sets condition code
- **BCR (07)**:
- **MR (1C)**:
- **DR (1D)**
- **LTR (12) (Load and test register)**: Sets condition code
- **LCR (13) (Load complement register)**: Sets condition code

  ```
  LCR   R1,R2
  ```

  If R2 contains n, R1 will contain $-n$
- **LNR (11) (Load negative register)**: Sets condition code

  ```
  LNR   R1,R2
  ```

  Causes the negative of the absolute value of R2 to be loaded into R1
- **LPR (10) (Load positive register)**: Sets condition code

  ```
  LPR   R1,R2
  ```

  Causes the absolute value of R2 to be loaded into R1. Overflow will occur if R2 contains the most negative number
- **BCTR (06) (Branch on count register)**: RR variant of BCT. Except, if R2 is R0, 1 is subtracted from R1, but no branch is taken

2.2 RX

- **A (5A)**: Sets condition code
- **S (5B)**: Sets condition code
- **L (58)**
- **ST (50)**
- **C (59)**: Sets condition code
- **BC (47)**
- **M (5C)**
- **D (5D)**
- **LA (41)**
- **BCT (07) (Branch on count)**:

  ```
  BCT   R,D(X,B)
  ```

  Causes contents of *R* to be decremented by one. If after this decrement *R* does not contain zero, a branch to the address D(X,B) is taken. If *R* does contain zero, the branch is not taken

- **CVB (4F) (RX) (Convert to binary)**: converts a packed decimal number in a doubleword of storage on a doubleword boundary to its binary equivalent and stores it in a register

  ```
  label CVB R1,D2(X2,B2) Explicit addr.
  label CVB R1,DWORD Implicit addr.
  ```

- **CVD (4E) (RX) (Convert to decimal)**: converts a binary number in the first operand register to its packed decimal equivalent in a doubleword on a doubleword boundary

  ```
  label CVD R1,D2(X2,B2) Explicit addr.
  label CVD R1,DWORD Implicit addr
  ```

- **BAL (45) (RX) (Branch and Link) Instruction**:

  ```
  label BAL R1,D2(X2,B2)
  ```

  - **Link Part of the Instruction:** Puts the last three bytes of the PSW into register R1 and zeros out the first byte of R1 . Why?

Remember that the last three bytes of the PSW hold the address of the next instruction, i.e., where to return to after the subroutine.

  - **Branch Part of the Instruction:** After the link part described immediately above, the BAL instruction then takes an unconditional branch to the D(X,B) address, i.e., the address of the subroutine.

2.3 SS (Storage to storage)

- **MVC (D2)**
- **CLC (D5)**: Sets condition code
- **Pack (F2) (SS) (Zoned to packed conversion)**: A little like XDECI, It translates numbers in character format into a format which can can be used for arithmetic

```
label PACK D1(L1,B1),D2(L2,B2)
label PACK label1(L1),label2(L2)
```

  PACK converts zoned decimal numbers into packed decimal numbers. Consider the zoned decimal F1F2C3, It does the following

  - The rightmost byte of the second operand is placed in the rightmost byte of the first operand, with zone and numeric digits reversed
  - The zone digits are stripped away and the remaining numeric digits from the second operand are moved to the first operand, right to left

- **UNPK (F3) (SS) (Packed to zoned conversion)**: A little like XDECO, It translates numbers in a format which can be used for arithmetic to numbers in character format

```
label UNPK D1(L1,B1),D2(L2,B2)
label UNPK label1(L1),label2(L2)
```

- **AP (FA) (SS) (Add)**: Add one packed decimal field to another

```
AP PFIELD1(5),PFIELD2(2)
```

- **SP (FB) (SS) (Subtract)**: Subtract one packed decimal field from another

```
SP PFIELD1(5),PFIELD2(2)
```

- **ZAP (F8) (SS) (Zero and add packed)**: Copy one packed decimal field to

```
ZAP PFIELD1(5),PFIELD2(2)
```

- **MP (FC) (SS) (Multiply)**: multiply one packed decimal field by another.

```
label MP D1(L1,B1),D2(L2,B2) Explicit addr.
label MP PNUM1(L1),PNUM2(L2) Implicit addr
```

- **DP (FD) (SS) (Divide)**: divide one packed decimal field by another

```
label DP D1(L1,B1),D2(L2,B2) Explicit addr.
label DP PNUM1(L1),PNUM2(L2) Implicit addr.
```

  Both quotient and remainder are stored in the first operand. Also does not set the condition code.

  Note: Remainder will be in the last *n* bytes, where *n* is the size of the second operand (divisor).

- **CP (F9) (SS) (Compare)**: compare one packed decimal field with another.

```
label CP D1(L1,B1),D2(L2,B2) Explicit addr.
label CP PNUM1(L1),PNUM2(L2) Implicit addr.
```

- **SRP (F0) (SS) (Shift and round)**: Shift a packed decimal by decimal digits left or right

```
label SRP D1(L,B1),D2(B2),i Explicit addr.
label SRP PNUM1(L),D2(B2),i Implicit addr.
```

- **ED (DE) (SS) (Edit)**: converts a packed decimal number to its printable EBCDIC equivalent.

```
label ED D1(L1,B1),D2(B2) Explicit addr.
label ED ONUM1(15),PNUM1 Implicit addr.
```

- **EDMK (DF) (SS)**: Edit and Mark (EDMK) is used exactly the same as Edit (ED). The only difference is that EDMK places the address of the first

2.4 SI (Storage to immediate byte)

- **MVI (92) (Move immediate)**: Replacement of the contents of the byte at D(B) with a copy of the immediate byte

```
MVI   D(B),byte
```

- **CLI (95) (Compare logical immediate)**: Sets condition code

  Compares the byte at D(B) with the specified byte

2.5 RS (Register to storage)

- **STM (90) (RS) (Store Multiple) Instruction**

```
label STM R1,R2,D(B)
```

Stores all registers from R1 through R2 to contiguous fullwords in that order in storage starting at D(B).

If R2 is less than R1 , then R1 through register 15 and register 0 through R2 are stored in that order in storage starting at D(B).

- **LM (98) (RS) (Load Multiple) Instruction**:

```
label LM R1,R2,D(B)
```

Loads all registers from R1 through R2 from contiguous fullwords in that order from storage starting at D(B).

If R2 is less than R1 , then R1 through register 15 and register 0 through R2 are loaded in that order from storage starting at D(B)

2.6 X instructions

- **XDUMP**

```
XDUMP   D(X,B),Length, Any comments, notice the comma
XDUMP                Dump it all
```

- **XREAD**

```
XREAD   D(X,B),length
```

Note that the length is usually 80

- **XDECI**

```
XDECI   R,addr
```

- **XDECO**

```
XDECO   R,addr
```

Note that XDECO requires character storage of length 12 bytes

- **XPRNT**

```
XPRNT   addr,len
```

Note that len is usually 133

Condition codes

- **Numeric and character compares**
  - **Zero** if $a = b$
  - **One** if $a < b$
  - **Two** if $a > b$
- **LTR**
  - **Zero**: If the loaded value is zero
  - **One** If the loaded value is negative.
  - **Two**: If the loaded value is positive
- **LCR**
  - **Zero**: If the loaded value is zero
  - **One** If the loaded value is negative.
  - **Two**: If the loaded value is positive
  - **Three:** If there is overflow
- **LNR**
  - **Zero**: If the loaded value is zero
  - **One**: If the loaded value is negative (R2 was nonzero)
- **LPR**
  - **Zero**: If the loaded value is zero
  - **Two**: If the loaded value is positive
  - **Three**: If overflow occurred
- **XREAD**
  - **Zero**: Read success
  - **One**: No more to read
- **XDECI**
  - **Zero**: Number was converted to zero
  - **One:** Number was converted less than zero
  - **Two** Number was converted greater than zero
  - **Three**: Tried to convert invalid number
- **Packed instruction condition codes**
  - **AP, SP, ZAP**:
    - 0 = the result is zero
    - 1 = the result is negative
    - 2 = the result is positive
    - 3 = overflow

Note: Note that overflow does not automatically occur if the first operand is shorter than the second, only when the result of the arithmetic operation does not fit into the first operand

- **CP**:
  - 0 = the two values compared are equal
  - 1 = the first operand's value is less than the second operand's
  - 2 = the first operand's value is greater than the second operand's
  - 3 - not used
- **SRP**:
  - 0 = result is zero
  - 1 = result is negative
  - 2 = result is positive
  - 3 = overflow has occurred
- **ED, EDMK**:
  - 0 = source field is zero
  - 1 = source field is negative
  - 2 = source field is positive
  - 3 = unused

Encodings

- **RR**

$$OP_1OP_2R_1R_2$$

- **RX**

$$OP_1OP_2R_1I_1B_1D_1D_2D_3$$

- **RS**:

$$h_0h_0h_{r_1}h_{r_2}h_Bh_Dh_Dh_D.$$

- **SS (Type 3)**

$$OP_1OP_2L_1L_2B_1DDDB_2DDD$$

  Note: L_1L_2 is the length minus 1

- **SS (Type 2)**:

$$h_0h_0h_{L_1}h_{L_2} \quad h_{B_1}h_{D_1}h_{D_1}h_{D_1} \quad h_{B_2}h_{D_2}h_{D_2}h_{D_2}.$$

  where

  - h_0h_0: Specifies the opcode
  - $h_{L_1}h_{L_2}$: Specifies the length-1 of the arguments lengths
  - $h_{B_1}h_{D_1}h_{D_1}h_{D_1}$: Address of the first operand
  - $h_{B_2}h_{D_2}h_{D_2}h_{D_2}$: Address of the second operand

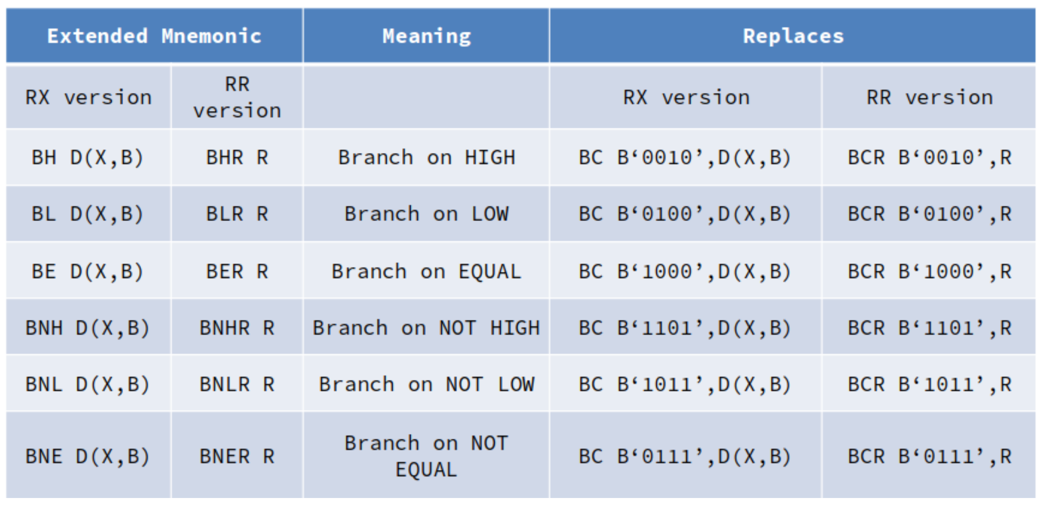
- **SI**

$$OP_1OP_2I_1I_2BDDD$$

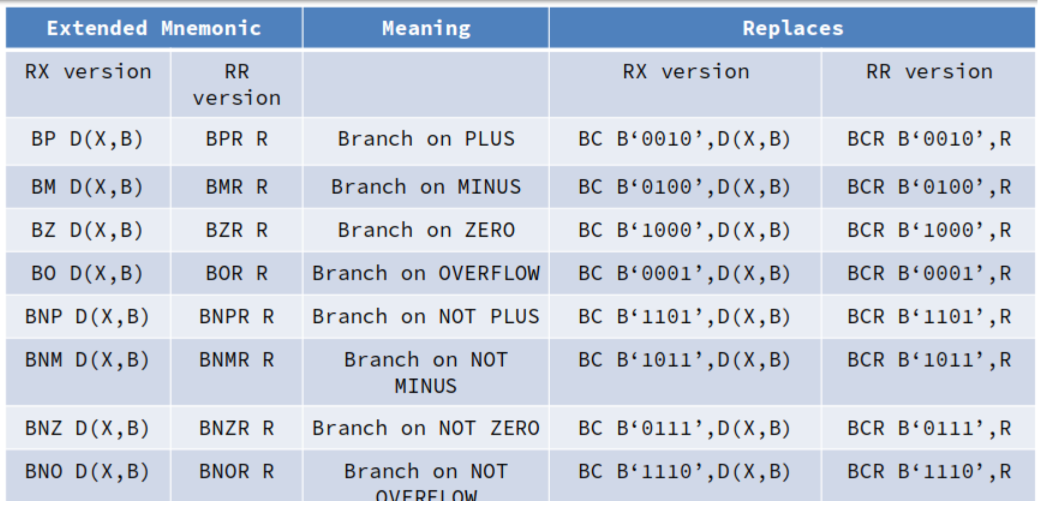
- **BCR B'1111',14**: Encoded as

$$07FE$$

Extended mnemonics



Extended Mnemonic		Meaning	Replaces	
RX version	RR version		RX version	RR version
BH D(X,B)	BHR R	Branch on HIGH	BC B‘0010’,D(X,B)	BCR B‘0010’,R
BL D(X,B)	BLR R	Branch on LOW	BC B‘0100’,D(X,B)	BCR B‘0100’,R
BE D(X,B)	BER R	Branch on EQUAL	BC B‘1000’,D(X,B)	BCR B‘1000’,R
BNH D(X,B)	BNHR R	Branch on NOT HIGH	BC B‘1101’,D(X,B)	BCR B‘1101’,R
BNL D(X,B)	BNLR R	Branch on NOT LOW	BC B‘1011’,D(X,B)	BCR B‘1011’,R
BNE D(X,B)	BNER R	Branch on NOT EQUAL	BC B‘0111’,D(X,B)	BCR B‘0111’,R



Extended Mnemonic		Meaning	Replaces	
RX version	RR version		RX version	RR version
BP D(X,B)	BPR R	Branch on PLUS	BC B‘0010’,D(X,B)	BCR B‘0010’,R
BM D(X,B)	BMR R	Branch on MINUS	BC B‘0100’,D(X,B)	BCR B‘0100’,R
BZ D(X,B)	BZR R	Branch on ZERO	BC B‘1000’,D(X,B)	BCR B‘1000’,R
BO D(X,B)	BOR R	Branch on OVERFLOW	BC B‘0001’,D(X,B)	BCR B‘0001’,R
BNP D(X,B)	BNPR R	Branch on NOT PLUS	BC B‘1101’,D(X,B)	BCR B‘1101’,R
BNM D(X,B)	BNMR R	Branch on NOT MINUS	BC B‘1011’,D(X,B)	BCR B‘1011’,R
BNZ D(X,B)	BNZR R	Branch on NOT ZERO	BC B‘0111’,D(X,B)	BCR B‘0111’,R
BNO D(X,B)	BNOR R	Branch on NOT OVERFLOW	BC B‘1110’,D(X,B)	BCR B‘1110’,R