Chapter 14: Data Comparison and Branching

The purpose of this chapter is to bring together a number of topics discussed in earlier chapters, so that we may emphasize a few of the more important points. As the chapter emphasizes comparison and branching, we begin with a theoretical discussion of sorting.

All comparison is based on the idea of a sort order. There are four possible operators that can be applied to give a sort order; these are usually denoted as "<", ">", "≤", and "≥". The two operators "=" and "≠" are interesting, but will not do for sorting.

As we can choose any of the four operators to support a sort, your author arbitrarily chooses the "less than" operator and, by implication, its opposite "greater than or equal".

A < B A is less than B. A precedes B in sort order.
A ≥ B A is not less than B. A does not precede B in sort order.

In order to support a sort order, a data type must have the following two properties.

1) Every pair of elements in the data type can be explicitly compared. Let A and B belong to the data type. Then, it is the case that either A < B or A ≥ B.
2) The transitivity property holds. Let A, B, and C belong to the data type. If A < B and B < C, it necessarily holds that A < C.

As an aside, it is easy to discover data types that do not suggest an explicit comparison. The easiest example is that of points in a plane, which are specified by two coordinates. Data types that always support explicit comparison but do not support the transitivity property can be devised, but they are more artificial.

The natural tendency would be to define two basic sort orders: numeric and alphabetic. While this works for data handled manually by humans, it is not really appropriate for data stored in computers. All data stored in computers, being binary, are ultimately subject to a numeric comparison. This statement holds for all primitive data types.

The numeric data types (halfword integer, fullword integer, packed decimal, and all floating point types) all support the expected sort order. Another way to put this is, that subject to precision and range limitations, the standard rules of algebra apply. The sort order for character data follows the EBCDIC format, which is ultimately an 8–bit integer format.

The reader should note that the EBCDIC sort order is equivalent to alphabetical order only when comparing characters all in upper case or all in lower case. There are a few surprises resulting from the decision to design EBCDIC for easy translation from punch card codes.

The following is a true sort sequence in EBCDIC: "a" < 'h" < "z" < "A" < "H" < "Z". The reason for this is seen in the following table of codes.

Character	"A"	"H"	"Z"	"a"	"h"	"z"
EBCDIC	**X'C1'**	**X'C8'**	**X'E9'**	**X'81'**	**X'81'**	**X'A9'**

The following is also a true sort sequence in EBCDIC: "I" < "!" < "J", as the hexadecimal codes are **X'C8', X'D0',** and **X'D1'**. In standard alphabetical order, there is no letter that stands between "I" and "J".

One may imagine a sort order on Boolean data, but it does not help very much. Boolean variables can have only two values 0 (False) or 1 (True). What we can clearly say is that $0 \neq 1$, or False≠ True. It is also commonly claimed that False < True, but this seem to your author to have little practical significance. In any case, we cannot have transitivity with only two values to consider. The reader should arrive at an independent opinion on this issue.

Branching Strategy

The essence of conditional branching is the two–step process of data–type specific comparison followed by a branch based on the results. There are a number of design strategies that can be used to design the appropriate instruction set. IBM seems to have selected the idea of a standard set of branch conditions (with synonyms) and designing each comparison instruction to set one of the branch conditions.

The reader should remember that the following instructions also set the branch conditions.

Binary arithmetic:	**A, S, AH, SH, AR, SR**
Boolean	**N, NC, NI, NR,** **O, OC, OI, OR,** **X, XC, XI, XR**
Packed Format:	**AP, SP, ZAP**

The conditional branch strategy operates based on the condition codes, which are derived from bits in the PSW (Program Status Word). For the System/370 these are bits 18 and 19 of the 64 bit PSW. Remember that bits are numbered left to right; the leftmost bit is bit 0.

The two bits stored in the PSW give rise to four standard condition codes. These are:

Condition Code		Interpretation
Binary	Decimal	
0 0	0	Result is zero or comparison is equal. For Boolean operators, the result has no bits with value 1.
0 1	1	Result is negative or the first argument sorts lower than the second. For Boolean operators, the result has at least one 1 bit.
1 0	2	Result is positive or the first argument sorts higher than the second.
1 1	3	Arithmetic overflow has occurred.

As noted previously, it seems a bit odd that the Boolean operator would set the negative bit when the result contains a 1 bit. Consider the following two binary values, each having 16 bits and possible to interpret as a halfword.

```
0011 1100 1101 1110   As a 16–bit integer, this is positive.
1111 1100 1101 1110   As a 16–bit integer, this is negative.
```

One might expect an integer interpretation of the results of the bitwise Boolean operations as implemented on the System/370. IBM must have had a good reason to implement the instructions this way; but all we say is “that is the way it was done”.

Branch Instructions

One of the encodings used to minimize the instruction size is to use the idea of a condition mask to extend two basic branch instructions into fourteen equivalent branch instructions. This device is often called "syntactic sugar" or extended mnemonics. There are two basic branch instructions in the IBM instruction set.

```
BC    MASK,TARGET      A TYPE RX INSTRUCTON
BCR   MASK,REGISTER    A TYPE RR INSTRUCTION
```

In the Type RX instruction, the target address is computed using the base register and displacement method, with an optional index register: **D2(X2,B2)**. In the Type RR instruction, the target address is found as the contents of the register.

Each of these instruction formats uses a four–bit mask, with bit numbers based on the 2–bit value of the condition code in the PSW, to determine the conditions under which the branch will be taken. The mask should be considered as having bits numbered left to right as 0 – 3.

Bit 0 is the equal/zero bit. Bit 2 is the high/plus bit.
Bit 1 is the low/minus bit. Bit 3 is the overflow bit.

The Standard Combinations

The following table shows the standard conditional branch instructions and their translation to the BC (Branch on Condition). The same table applies to BCR (Branch on Condition, Register), so that there is another complete set of mnemonics for that set.

Bit Mask Flags				Condition		Extended instructions	
0	1	2	3			Sort	Arithmetic
0	0	0	0	No branch	BC 0,XX	NOP	
0	0	0	1	Bit 3: Overflow	BC 1,XX	BO XX	
0	0	1	0	Bit 2: High/Plus	BC 2,XX	BH XX	BP
0	1	0	0	Bit 1: Low/Minus	BC 4,XX	BL XX	BM
0	1	1	1	1, 2, 3: Not Equal	BC 7,XX	BNE XX	BNZ
1	0	0	0	Bit 0: Equal/Zero	BC 8,XX	BE XX	BZ
1	0	1	1	0, 2, 3: Not Low	BC 11.XX	BNL XX	BNM
1	1	0	1	0, 1, 3: Not high	BC 13,XX	BNH XX	BNP
1	1	1	1	0, 1, 2, 3: Any	BC 15,XX	B XX	

Note the two sets of extended mnemonics: one for comparisons and an equivalent set for the results of arithmetic operations.

These equivalent sets are provided to allow the assembler code to read more naturally.

Here is the complete listing of S/370 assembler mnemonics, showing what is called either "extended mnemonics" or "syntactic sugar" for each of the BC and BCR instructions.

	Branch on Condition (Register)			Branch on Condition		
Condition	Basic	Extended Instruction		Basic	Extended Instruction	
		Sort	Arithmetic		Sort	Arithmetic
No branch	BCR 0,R	NOPR		BC 0,X	NOP	
Bit 3: Overflow	BCR 1,R		BOR R	BC 1,X		BO X
Bit 2: High/Plus	BCR 2,R	BHR R	BPR R	BC 2,X	BH X	BP X
Bit 1: Low/Minus	BCR 4,R	BLR R	BMR R	BC 4,X	BL X	BM X
1, 2, 3: Not Equal	BCR 7,R	BNER R	BNZR R	BC 7,X	BNE X	BNZ X
Bit 0: Equal/Zero	BCR 8,R	BER R	BZR R	BC 8,X	BE X	BZ X
0, 2, 3: Not Low	BCR 11.R	BNLR R	BNMR R	BC 11.X	BNL X	BNM X
0, 1, 3: Not high	BCR 13,R	BNHR R	BNPR R	BC 13,X	BNH X	BNP X
0,1, 2 No overflow	BCR 14,R		BNOR R	BC 14,X		BNO X
0, 1, 2, 3: Any	BCR 15,R	BR R		BC 15,X	B X	

The BC Instruction

The BC instruction is a type RX instruction with opcode **X'47'**.
The source code format for the instruction is **BC M,Address** or **BC M,D2(X2,B2).**
The object code format for this instruction is as follows.

Type	Bytes	Operands	1	2	3	4
RX	4	R1,D2(X2,B2)	**47**	M_1 X_2	B_2 D_2	D_2D_2

The first byte contains the 8–bit instruction code, which is **X'47'**.

The second byte contains two 4–bit fields.

The first four bits contain the mask for the branch condition codes
The second four bits contain the number of the index register used in computing the address of the jump target. If $X_2 = 0$, indexed addressing is not used.

The next two bytes contain the 4–bit number of the base register and the 12–bit displacement used to form the basic address of the target. This address may be indexed.

Suppose that address **TARGET** is formed by offset **X'666'** using base register 8.
No index is used and the instruction is **BNE TARGET**, equivalent to **BC 7,TARGET**,
as the condition mask for "Not Equal" is the 4–bit number **0111**, or decimal 7.

The object code for this is **47 70 86 66.**

The BCR Instruction
The BCR instruction is a two–byte instruction of the form **`OP M1,R2`**.

Type	Bytes	Operands		
RR	2	M1,R2	07	M_1 R_2

The first byte contains the 8–bit instruction code, which is **X'07'**.

The second byte contains two 4–bit fields.

The first 4–bit field encodes a branch condition

The second 4–bit field encodes the number of the register containing the target address for the branch instruction.

For example, the instruction **`BR R8`** is the same as **`BCR 15,R8`**. Again, source code written in this form assumes that the symbol **`R8`** has been defined with an **`EQU`** directive.

The object code is **`07 F8.`** The effect of this instruction is to branch unconditionally to the address in register 8. An instruction such as this is used to implement a subroutine return.

The two code fragments below may be considered to be equivalent. The first fragment is:

```
LA R8,TARGET    Load R8 with the target address
BR R8           Branch to the address in R8.
```

The second fragment avoids the register as follows:

```
B  TARGET       Branch to the target address.
```

Comparing Binary Data: C, CH, and CR
There are three instructions for binary comparison with the value in a register.

Mnemonic	Description	Type	Format
C	Compare full–word	RX	**`C R1,D2(X2,B2)`**
CH	Compare half–word	RX	**`CH R1,D2(X2,B2)`**
CR	Compare register to register	RR	**`CR R1,R2`**

Each comparison sets the expected condition code.

Condition	Condition Code	Branch Taken
Operand 1 = Operand 2	0 (Equal/Zero)	BE, BZ
Operand 1 < Operand 2	1 (Low/Minus)	BL, BM
Operand 1 > Operand 2	2 (High/Plus)	BH, BP

Each of the **C** and **CH** instructions is a type RX instruction, with source code shown above. The object code is of the form that is standard for a type RX instruction.

Type	Bytes	Operands	1	2	3	4
RX	4	R1,D2(X2,B2)	OP	R_1 X_2	B_2 D_2	D_2D_2

The first byte contains the 8–bit instruction code. The opcode for C is **X'59**' and that for CH is **X'49**'. The second byte contains two 4–bit fields, each of which encodes a register number. The first hexadecimal digit specifies the register containing either the fullword or the halfword that will be compared. The second hexadecimal digit specifies the optional register that may be used if indexed addressing is used. If this digit is 0, then the addressing is simple base/displacement with out any addressing.

The next two bytes contain the 4–bit number of the base register and the 12–bit displacement used to form the basic address of the target. This address may be indexed.

For the **CH** (Compare Halfword) instruction, the address computed will reference a halfword. The value in this halfword will be sign extended to a 32–bit fullword before the comparison.

The CR instruction is a type RR instruction, with opcode **X'19'**.
This is a two–byte instruction of the form **OP R1,R2**.

Type	Bytes	Operands		
RR	2	R1,R2	OP	R_1 R_2

The first byte contains the 8–bit instruction code.
The second byte contains two 4–bit fields, each of which encodes a register number.

Don't forget that literal arguments can be used with either **C** or **CH**, as in this example.

```
         C  R9,=F'0'   COMPARE THE REGISTER TO ZERO
         BH ISPOS      IT IS POSITIVE
         BL ISNEG      NO, IT IS NEGATIVE.
         If this line is reached, R9 contains the value 0.
         More code here

         B  DOMORE
ISPOS    Code for R9 > 0 is placed here

         B  DOMORE
ISNEG    Code for R9 < 0 is placed here

DOMORE   Common code is placed here.
```

CP: Compare Packed Decimal Fields

The **CP** instruction is a type SS instruction with opcode **X'F9'**. The source code is of the form **OP D1(L1,B1),D2(L2,B2)**, which provide a 4–bit number representing the length for each of the two operands. The object code format is as follows.

Type	Bytes	Operands	1	2	3	4	5	6
SS(2)	6	D1(L1,B1),D2(L2,B2)	**X'F9'**	L_1 L_2	B_1 D_1	D_1D_1	B_2 D_2	D_2D_2

The first byte contains the operation code, which is **X'FA'**.
The second byte contains a two values, each a 4–bit binary number (one hex digit).

L1 A value that is one less than the length of the first operand.
L2 A value that is one less than the length of the second operand.

Bytes 3 and 4 specify the address of the first operand, using the standard base register and displacement format. Bytes 5 and 6 specify the address of the second operand, using the standard base register and displacement format.

Some of the rules for the **CP** instruction are as follows.

1. The maximum field length for either operand is 16 bytes; thus 31 digits.
2. Both operands must contain valid packed data.
3. If the fields are not the same length, CP extends the shorter field (in the CPU, but not the copy in memory) by padding it with the proper number of leftmost zeroes before comparison.
4. The CP instruction follows the rules of standard algebra. Any positive number will sort greater than any negative number, except that +0 = –0.
5. The implicit number of decimal places in each operand must be the same, or the comparison results will not reflect the true meaning of the data.

Example

Consider the assembly language statement below, which compares **AMOUNT** to **TOTAL**.

```
CP TOTAL,AMOUNT
```

Assume:
1. **TOTAL** is 4 bytes long, so it can hold at most 7 digits.
2. **AMOUNT** is 3 bytes long, so it can hold at most 5 digits.
3. The label **TOTAL** is at an address specified by a displacement of **X'50A'** from the value in register **R3**, used as a base register.
4. The label **AMOUNT** is at an address specified by a displacement of **X'52C'** from the value in register **R3**, used as a base register.

The object code looks like this: **F9 32 35 0A 35 2C**

The Disassembly of the Above Example
Consider **F9 32 35 0A 35 2C.** The operation code **X'F9'** is that for the Compare Packed (CP) instruction, which is a type SS(2). The above format applies.

The field **32** is of the form L_1 L_2.
The first value is **X'3'**, or 3 decimal. The first operand is 4 bytes long.
The second value is **X'2'**, or 2 decimal. The second operand is 3 bytes long.

The two–byte field **35 0A** indicates that register 3 is used as the base register for the first operand, which is at displacement **X'50A'**.

The two–byte field **35 2C** indicates that register 3 is used as the base register for the second operand, which is at displacement **X'52C'**.

It is quite common for both operands to use the same base register.

Character Comparison: CLC
The **CLC (Compare Logical Character)** instruction is one of the two used to compare character fields, one byte at a time, left to right. The instruction is type SS with opcode **X'D5'**. The source code is of the form **CLC D1(L,B1),D2(B2)**, which provide a length for only operand 1. The length is specified as an 8–bit byte. The object code format is:

Type	Bytes	Operands	1	2	3	4	5	6
SS(1)	6	D1(L,B1),D2(B2)	**X'D5'**	L	B_1 D_1	D_1D_1	B_2 D_2	D_2D_2

The first byte contains the operation code, which is **X'D5'** for **CLC**.

The second byte contains a value storing one less than the length of the first operand, which is the destination for any move. Bytes 3 and 4 specify the address of the first operand, using the standard base register and displacement format. Bytes 5 and 6 specify the address of the second operand, using the standard base register and displacement format.

Comparison is based on the binary contents (EBCDIC code) contents of the bytes.
The sort order is from X'00' through X'FF'.
The instruction may be written as **CLC Operand1,Operand2**

An example of the instruction is **CLC NAME1,NAME2**

This instruction sets the condition code that is used by the conditional branch instructions. The condition code is set as follows:

If Operand1 is equal Operand2 Condition Code = 0

If Operand1 is lower than Operand2 Condition Code = 1

If Operand1 is higher than Operand2 Condition Code = 2

The operation moves, byte by byte, from left to right and terminates as soon as an unequal comparison is found or one of the operands runs out.

Example of Character Instructions
Consider the example assembly language statement, which compares the string of characters at label **CONAME** with those at the location associated with the label **TITLE**.

```
CLC TITLE,CONAME
```

Suppose that:
1. There are fourteen bytes associated with **TITLE**, say that it was declared as **TITLE DS CL14**. Decimal 14 is hexadecimal E.
2. The label **TITLE** is referenced by displacement **X'40A'** from the value stored in register **R3**, used as a base register.
3. The label **CONAME** is referenced by displacement **X'42C'** from the value stored in register **R3**, used as a base register.

Given that the operation code for CLC is **X'D5'**, the instruction assembles as
```
D5 0D 34 0A 34 2C   Length is 14 or X'0E'; L – 1 is X'0D'
```

Final Comments on the Comparison Operators
This chapter has covered a variety of typical comparison operators.
C, CH, and CR for integer comparison,
CP for packed decimal comparison, and
CLC for character comparison.

At this point, it would be helpful to restate the warning that it is the instruction that sets the data type for the section of memory that is being referenced. If the data are of the wrong type for the instruction, strange results may be produced.

For example, suppose that the labels DATA1 and DATA2 have been defined somehow and that they have been initialized with some content.

The instruction **C R6,DATA1** will compare the contents of register R6 with the contents of the four bytes beginning at address DATA1, assuming that those four bytes represent a 32–bit integer in two's–complement form.

The instruction **CH R6,DATA1** will compare the contents of register R6 with the sign extended contents of the two bytes beginning at address DATA1, assuming that those two bytes represent a 16–bit integer in two's–complement form.

The instruction **CP DATA1,DATA2** will compare the contents of the two locations, assuming that each location contains data in valid packed decimal format.

The instruction **CLC DATA1,DATA2** will compare the contents of the two locations, assuming that each location contains EBCDIC character data.

For data that are properly initialized (this is a bit of a trick), it might be possible that each of the four instructions will execute to completion within the same program. All four might give results, but only one will give a correct result.