Chapter 9: Bin There, Done That…

In which we learn to add, subtract, multiply and divide binary data.

Defining Binary Data

Fixed-point data fields, more commonly called "Binary" data fields, are represented in a 2's complement format that allows integers to be positive or negative. Binary fields are defined using an F or an H as the data type in a DS or DC declarative. Some binary operations require 8-byte doublewords that are defined using D as the data type, but D is technically a floating-point designator. Here are a few examples:

```
A          DC    F'20'
B          DC    H'-30'
C          DC    F'0'
```

By default, the F data type creates a fullword (4-byte) field containing a binary integer in 2's complement format. Each field is automatically aligned on a fullword boundary (an address that is divisible by 4). Fullword values range from -2,147,483,648 to +2,147,483,647 (Think plus or minus 2 gigs.)

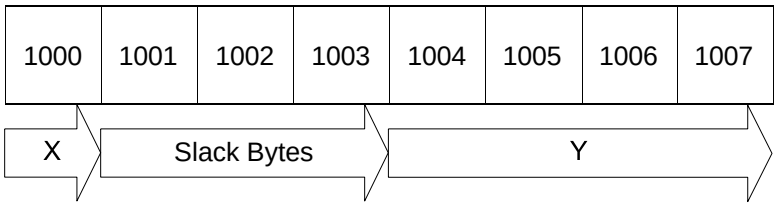
The H designator is used to create halfword (2-byte) fields containing 2's complement signed integers that are aligned on a halfword boundaries (addresses that are divisible by 2). Halfword values range from -32,768 to +32,767 (Think plus or minus 32,000.) Here are two examples:

```
D          DS    H'32'
E          DS    H'-7'
```

In both formats, the length indicator Ln is usually omitted since the length is understood to be 2 or 4, but values from 1 to 8 are possible. When the length indicators are absent, the fields will be automatically aligned on either halfword or fullword boundaries. To align a field properly, the assembler may generate from 1 to 7 unused "slack" bytes. Consider the example below. We assume that field "X" is located at address x'1000' and that the addresses of the bytes are shown inside each byte.

LOCATION

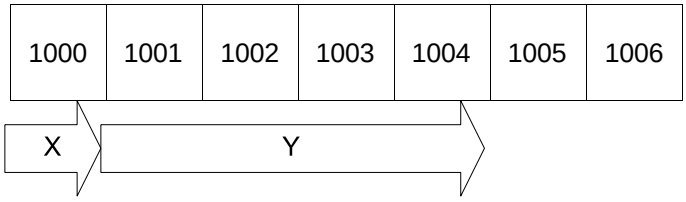
```
1000  X       DS  CL1
1004  Y       DS  F
```



It is possible to code a length from 1 to 8 for fullword and halfword binary fields. For example, we might code FL4 or HL2. When a length is indicated, slack bytes will not be generated, and the field is not automatically aligned. Consider the following example, which is similar to the example above, but generates no slack bytes.

LOCATION

```
1000  X      DS     CL1
1004  Y      DS     FL4
```



Fullword and halfword alignment can also be obtained with another technique: Coding a 0 for the duplication factor forces proper alignment for the succeeding field. For instance, the following example forces FIELDA to be fullword aligned.

```
            DS  0F
FIELDA      DS  ...
```

Examples

Some Typical DS's and DC's:

```
A     DS   F              A FULLWORD FIELD, PROPERLY ALIGNED
B     DS   H              A HALFWORD FIELD, PROPERLY ALIGNED
C     DS   FL4            A FULLWORD FIELD, NO SLACK BYTES GENERATED
D     DS   FL4            A HALFWORD FIELD, NO SLACK BYTES GENERATED
E     DC   F'2147483647'  MAXIMUM VALUE OF A FULLWORD (2^31) -1
F     DC   H'32767'       MAXIMUM VALUE OF A HALFWORD (2^15) -1
G     DC   F-2147483648'  MINIMUM VALUE OF A FULLWORD -(2^31)
H     DC   H'-32768'      MINIMUM VALUE OF A HALFWORD -(2^15)
I     DC   F'0'           A FULLWORD ZERO
J     DC   H'40000'       ASSEMBLY ERROR - CONSTANT TOO LARGE
K     DS   0F             PROVIDE FULLWORD ALIGNMENT FOR NEXT FIELD
L     DS   0H             PROVIDE HALFWORD ALIGNMENT FOR NEXT FIELD
M     DC   H'20           M = X'0014'     2'S COMPLEMENT INTEGER
N     DC   F'-20'         N = X'FFFFFFEC' 2'S COMPLEMENT INTEGER
O     DC   H'92'          O = X'005C'     2'S COMPLEMENT INTEGER
```

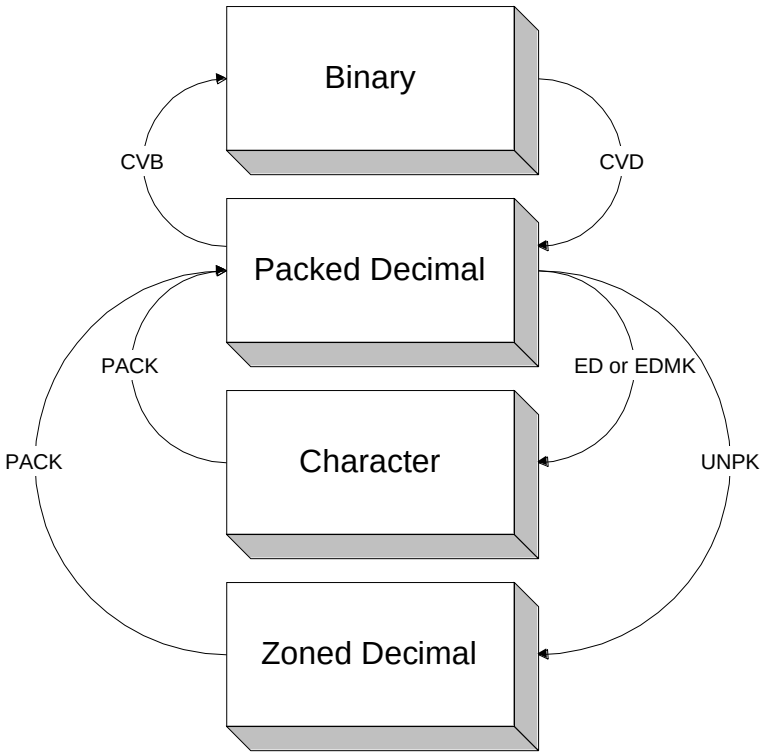
Tips

1) The contents of halfword fields are small - between 32767 and -32768. Memory is cheap, so think twice before using halfword fields.

2) Remember to code the length indicator in cases where you don't want slack bytes generated to guarantee alignment.

Changing Data Types

The diagram below shows the instructions used to convert between four important data types. In this chapter we consider the conversion instructions used between binary and packed decimal data.



Converting from Packed Decimal to Binary

In converting to binary with the **CVB** instruction, there is a requirement that the packed field that is to be converted must be stored in a properly aligned doubleword. Not all packed decimal values can be converted to binary in a register because of the limitations of a 32-bit register. The range of integers which can be successfully converted is -2,147,483,648 to +2,147,483,647. Since the packed fields we work with are rarely 16 bytes long, many programmers will use a doublword work area as a staging area for the conversions. Simply **PACK** or **ZAP** the doubleword with the required field before converting to binary in a register:

```
          ZAP   DBLWD,PFIELD  TRANSFER FIELD TO DBLWD STAGING AREA
          CVB   R6,DBLWD      CONVERT TO BINARY IN R6
          ...
PFIELD    DC    P'2345'
DBLWD     DS    D             DOUBLE WORD STAGING AREA
```

In the example above PFIELD contains a packed decimal value and we wish to convert this value to 2's complement binary in a register. First, we move the field to a double word staging area called DBLWD. Since PFIELD is packed, the field is transferred to DBLWD with a **ZAP** rather than an **MVC**. The **CVB** instruction performs the conversion from packed to binary.

Converting from Binary to Packed Decimal

The requirements for converting from binary to packed decimal using the **CVD** instruction are like those for using **CVB**. In this case, the field that is to receive the packed decimal field must be a doubleword. Since a doubleword can hold up to 15 packed digits, **any** value in a register can be converted to packed decimal. A small problem arises after the conversion if we wish to edit the data into a printable format. Since the doubleword will contain 15 decimal digits, editing the result requires a large edit word that can be a bit cumbersome. A common practice is to **ZAP** the double word into a smaller packed field that could hold the result before proceeding to edit the data:

```
          CVD   R6,DBLWD          CHANGE TO DECIMAL
          ZAP   SMALLPK,DBLWD     RESIZE TO A SMALLER FIELD
                ...
DBLWD     DS    D
SMALLPK   DS    PL4
```

The size of the smaller packed field is somewhat arbitrary. It must be big enough to hold the maximum integer you might process. Know your data!

A Complete Example

The following code demonstrates how a single zoned-decimal field (ZFIELD) can be converted to three other data types.

```
                PACK  PKFIELD,ZFIELD  NOW WE HAVE A PACKED VERSION
                ZAP   DBLWD,PKFIELD   PREPARE TO CONVERT TO ...
                CVB   R8,DBLWD        ...A BINARY VERSION IN R8
                MVC   CHOUT,EDWD      GET READY TO EDIT...
                ED    CHOUT,PKFIELD   ... A CHARACTER VERSION IN CHOUT
                UNPK  ZOUT,PKFIELD    CONVERT BACK TO ZONED IN ZOUT
                ...
        DBLWD   DS    D
        ZFIELD  DS    ZL5             5 ZONED DIGITS
        PKFIELD DS    PL3             3 BYTES WILL HOLD ZFIELD'S DATA
        EDWD    DC    X'40202020212060'  7-BYTE EDIT WORD
        CHOUT   DS    CL7
        ZOUT    DS    ZL5
```

Binary Arithmetic

Binary arithmetic occurs in the general-purpose registers with the data in a two's complement format. Some arithmetic operations occur between registers (RR) as in the Add Register instruction below:,

```
         AR    R5,R6
```

This operation adds the contents of R6 to the contents of R5, leaving the sum in R5. R6 remains unchanged. Other binary operations occur between a register and a storage location (RX), as in the following Subtract Fullword instruction:

```
         S     R5,NOITEMS
         ...
NOITEMS  DC    F'100'
```

This operation subtracts the fullword called NOITEMS from the fullword in R5, leaving the difference in R5. In RX instructions, the first operand is always a register and the second is a storage location denoted by a base register, an index register, and a displacement. Data may flow from operand 2 to operand 1 or vice versa depending on the operation. In the case of a subtract (S), the data flows from operand 2 and changes operand 1.

Because Subtract Fullword is an RX instruction, it might have been coded explicitly like this:

```
         S    R5,0(R3,R4)
```

The target is register 5, while the effective address of the source fullword is determined by adding the contents of base register 4, the contents of index register 3, and the 0 displacement. The term "index register" refers to the idea that the effective address is determined by adding a third component – the index - to the base/displacement address. This is similar to using a subscript (or index) with an array in a high-level language. Both base and index registers are simply general-purpose registers. **There is one important convention that you should note: When 0 is used as a base or index register, that part of the address is ignored.** For example, in the instruction below, the source address is simply the contents of register 6 plus an 18-byte displacement.

```
         S    R5,18(R0,R6)
```

Unlike SS instructions in which the target is always operand 1 and the source is always operand 2, the source and target for RX instructions depends on the instruction. In the Subtract Fullword instructions above, the target is operand 1 and the source is operand 2, but the situation is reversed for the Store Fullword instruction below:

```
         ST   R5,18(R0,R6)
```

The Store instruction copies the fullword in the operand 1 register to the storage location denoted by operand 2.

There are several characteristics of binary arithmetic which make it easier (in some respects) than packed decimal arithmetic.

1) The field sizes are usually uniform in size - 2, 4, or 8 bytes (mostly 4) and the instructions are tailored to these sizes. For example, the Subtract Fullword instruction

above is designed to work on fullwords.

2) We are usually working with pure integers which don't have implied decimals. When we convert packed decimal fields to binary, the packed-decimal numbers are treated as pure integers, so the conversions are exact.

Trying to convert packed decimal integers with implied decimals to binary representations is not usually a good idea because it involves a loss of precision. The reason for this is that base 10 is not an exact power of base 2. In representing numbers in base 10, the "weight" of each digit is a power of 10: …, 10^3, 10^2, 10^1, 10^0=1, 10^{-1}=1/10, 10^{-2} = 1/100, … Similarly, in base 2 the weights are …, 2^3, 2^2, 2^1, 2^0=1, 2^{-1}=1/2, 2^{-2}=1/4, … If we had infinitely many weights, any number represented in base 10 would have an exact representation in base 2, but on a computer, the number of digits we can devote to any representation is finite, so there is often a loss of precision when converting between these two bases.

3) Binary arithmetic occurs in the general-purpose registers, and if the program abends, we automatically get a listing of the contents of the registers.

There is one characteristic of binary arithmetic which makes it harder than packed arithmetic – two's complement integers aren't as easy to decipher as packed decimal integers. A programmer's calculator will help with this, but you still need a few basic facts and skills to understand how to use the calculator, make life simpler, and avoid some grief.

1) Two's complement integers come in fixed sizes – If you are working with fullwords, the integers are in 32-bit two's complement, while halfwords are 16-bit two's complement. Double words contain 64-bit two's complement integers. In doing arithmetic, you must determine what size you want to work with. Most often, you will be working with fullwords.

2) The high-order bit is the sign – 0 is positive and 1 is negative. We usually represent binary fields with hexadecimal digits. In this case, which hexadecimal digits represent binary numbers that have a 0 in the high-order bit? Answer: 0, 1, 2, …, 7. Which hexadecimal digits represent binary numbers that have a 1 in the high-order bit? Answer: 8, 9, A, B, C, D, E, and F. If we look at a two's complement integer represented in hex, you might not recognize the number immediately, but you will know if it is positive or negative by examining the high-order hex digit. Here are some examples:

```
x'73F9'     positive
x'FFFFFFFF' negative
x'C8B'      negative
x'00000F8'  positive
x'3456789'  positive
```

3) **Sign extending a two's complement integer leaves its value unchanged** - The representation size is changing whenever you sign extend, but not the value of the integer. Here are a few examples

00111 is the integer 7 in 5-bit two's complement
000111 is the integer 7 in 6-bit two's complement
0000111 is the integer 7 in 7-bit two's complement
11111 is the integer -1 in 5-bit two's complement
111111 is the integer -1 in 6-bit two's complement
1111111 is the integer -1 in 7-bit two's complement
11111111111111111111111111111111 is the integer -1 in 32-bit two's complement

4) **Determining the value of a two's complement integer** – In two's complement, positive integers are easier to read than negative integers. If the integer has a 0-sign bit (it's positive) simply treat the integer as a plain binary integer. For example, the two's complement integer 001101 represents +13. There's no difference in how positive integers are represented in plain binary than in two's complement. Negative two's complement integers are a bit harder to interpret. The easiest approach for evaluating negative integers is to compute the additive complement of the integer and then read that number in plain binary. For example, the additive complement of -3 is 3, and the additive complement of -5 is 5. Usually, when we are working with two's complement integers, the numbers are represented in hexadecimal to keep the representations somewhat readable. For example, x'FA4' is a kind of shorthand for b'111110100100', which is -92 in decimal, but that's hard to see directly. First we need to compute the additive complement of x'FA4' which is easier to read.

To compute a two's complement, follow this rule:

Change the binary 1's to 0's, change the 0's to 1's, and then add 1.

Let's try that for -92.

In binary:	111110100100	= -92
Flipping the 1's and 0's:	000001011011	
Adding 1:	000001011011	
	+ 1	
Two's complement:	000001011100	= +92

Let's compute the additive two's complement of +9 in 6-bits.

In binary:	001001	= +9
Flipping the 1's and 0's:	110110	
Adding 1:	110110	
	+ 1	
Additive Two's complement:	110111	= -9

5) There is a limit to what can be computed in any two's complement representation

-Here is a list of all the 3-bit binary patterns and their interpretations in binary and in two's complement.

Plain Binary	Two's Complement
000 = 0	000 = 0
001 = 1	001 = 1
010 = 2	010 = 2
011 = 3	011 = 3
100 = 4	100 = -4
101 = 5	101 = -3
110 = 6	110 = -2
111 = 7	111 = -1

This pattern is worth thinking about, as it similar to two's complement patterns for other sizes. The binary patterns on the left are in sequence if you think of the numbers as plain binary integers. The second column is the value of the same binary patterns interpreted as two's complement integers. In the right column, the numbers start at 0 and continue up to the largest positive integer 3. At that point, the next integer is the smallest integer that can be represented in 3-bit 2's complement, -4, and continues in sequence up to the largest negative integer that can be formed, -1.

Using numbers in this 3-bit representation allows for binary arithmetic:

```
    101 = -3
+   010 = +2
Result: 111 = -1
```

I'll point out three other facts about numbers expressed in two's complement:

a. -1 is always represented by all 1's, no matter what the representation size.

b. The smallest integer in an n-bit representation is represented by a 1 followed by all 0's and is equal to $-2^{(n-1)}$.

c. In any representation, there is one more negative integer than there are positive integers since 0 is represented as all 0's (in other words, zero has a positive sign).

Computations in every representation are limited by the number of bits. For example, if we try to add 3 + 3 in 3-bit two's complement, we can't possibly compute the correct answer because there is no representation for 6 using 3 bits. Nevertheless, an answer is produced.

```
    011 = +3
+   011 = +3
    110 = -2  Overflow!
```

This computation overflowed the representation size and produced an incorrect answer. For any fixed-size representation, overflow, and its companion – underflow,is a possibility. There is a rule concerning the sign bit that the hardware uses to determine

if a binary computation is correct. The computer keeps up with carries into and out of the sign bit. Valid computations occur when there are *no* carries into or out of the sign bit, and for computations with carries into *and* out of the sign bit. All other cases result in incorrect results. In the overflow above, there was a carry into the sign bit, but not out of the sign bit, so the result is incorrect.

Moving Data into and Out of the Registers

Copying the contents of a register into memory is referred to as "storing" and copying data from a storage location in memory into a register is usually called "loading". There is a plethora of load instructions and many fewer store instructions. The workhorse instructions in this group are Load Fullword (**L**) and Store Fullword (**ST**). Consider the following example where we want to add the contents of X and Y, leaving the sum in Z:

```
            L      R5,X      PREPARE TO ADD X
            L      R6,Y      ...AND Y
            AR     R5,R6     COMPUTE X + Y
            ST     R5,Z      PUT THE SUM IN Z
            ...
X           DC     F'8'
Y           DC     F'12'
Z           DS     F         F CONTAINS X'00000014'
```

The first load initializes R5 with X, the second initializes R6 with Y. We compute the sum by invoking Add Register (**AR**) in which the source is R6 and the target is R5. Add Register is the Register to Register (**RR**) version of the RX instruction, Add Fullword. The computation above is completed by using Store Fullword (**ST**) to copy the sum into Z.

Another fact that makes binary arithmetic easy is that by mastering one instruction, you will learn other related instructions by association. For example, Add Fullword, Add Register, and Add Halfword are all closely related instructions.

Here is an alternative computation that arrives at the same result, but uses one fewer register:

```
            L      R5,X      PREPARE TO ADD X
            A      R5,Y      ...and Y
            ST     R5,Z      PUT THE SUM IN Z
            ...
X           DC     F'8'
Y           DC     F'12'
Z           DS     F         F CONTAINS X'00000014'
```

There are two other instructions closely related to **L** and **ST** that move data between memory and registers: Load Multiple (**LM**) and Store Multiple (**STM**). These will be discussed in more detail when we discuss linkage, but for the moment, as the names imply, these instructions load or store the contents of multiple (consecutive) registers.

```
          LM    R3,R5,X     LOAD REGISTERS R3,R4,R5
*                           ... WITH X,Y,Z
          A     R3,=F'1'    INCREMENT R3
          A     R4,=F'1'    INCREMENT R4
          A     R5,=F'1'    INCREMENT R5
          STM   R3,R5,X     STORE RESULTS IN X,Y,Z
          ...
X         DC    F'8'
Y         DC    F'12'
Z         DS    F'20'
```

The **LM** instruction loads consecutive fullwords X, Y, and Z into consecutive registers R3, R4, and R5. The **STM** instruction stores the results into consecutive fullword X, Y, and Z

Patterns of Binary Arithmetic Instructions

For all addition and subtraction operations, a fullword is added to or subtracted from a register and the result left in a register. In the case of Add Fullword, the fullword is in memory. In the case of Add Register, the fullwords are in registers. In the case of Add Halfword, a halfword in memory is internally sign-extended to obtain a fullword which is then added to a register. The original halfword is unchanged. The same idea applies to Load Halfword, which takes a halfword in memory, sign-extends it to 32 bits, and copies the resulting fullword into a register. A Load Register (**LR**) instruction copies a fullword from one register to another. Load and Test Register (**LTR**) works identically to **LR** but also sets the condition code to indicate how the target register contents compare to zero. In the case of Store Halfword, the rightmost 16 bits (a halfword) of a register are stored in a halfword memory location.

You should now have a sense of how the following instructions are related and can be used.

RX	RX	RR
A	AH	AR
S	SH	SR
L	LH	LR, LTR
ST	STH	--

You might guess that similar operations exist for multiplying and dividing, as well as comparing, and you're right. Here's what's available:

RX	RX	RR	
M	MH	MR	Multiplications
D	--	DR	Divisions
C	CH	CR	Comparisons

Binary Multiplication

As in the case of packed decimal arithmetic, binary arithmetic for multiplying and dividing is slightly more complicated than for adding and subtracting. Let's consider an example of each. First, let's multiply fullwords X and Y below, leaving the product in Z. The first thing we must consider is whether the product will fit in the fullword Z. Let's assume it will.

```
            L     R7,X
            M     R6,Y
            ST    R7,Z
            ...
X           DC     F'5'
Y           DC     F'4'
Z           DS     F
```

After loading R7 with X, why did we multiply with R6 and not R7? The reason is that multiplication and division occur in pairs of Even/Odd consecutive register pairs like R6 and R7, or R10 and R11. The reason we need to use two registers for multiplying is that whenever you multiply two fullwords together, the result could be as large as a doubleword – this would require the product to occupy two registers. The before and after pictures of a multiplication (**M**) looks like the diagram below.

Before: Even Register: (uninitialized) Odd Register: (multiplicand)
After: Even Register: (High 32 bits of product) Odd Register: (Low 32 bits of product)

So, after the load and the before the multiplication above, the pictures might look like this with a random value in R6.

Before: Even Register 6: (x'F003ACD6') Odd Register 7: (x'00000005')
After: Even Register 6: (x'00000000') Odd Register 7: (x'00000014')

Here is a slightly different example using the same code. We are computing the product of X and Y, and storing the result in Z, showing the registers before and after the multiplication.

```
            L      R7,X
            M      R6,Y
            ST     R7,Z
            ...
X           DC     F'-3'
Y           DC     F'2'
Z           DS     F
```

Before: Even Register 6: (x'F003ACD6') Odd Register 7: (x'FFFFFFFD')
After: Even Register 6: (x'FFFFFFFF') Odd Register 7: (x'FFFFFFFA')

After the multiplication, the even/odd register combination contains a 64-bit result. In most cases, the product can be taken from the odd register and the value in the even register (all zeroes or all ones) is ignored. This occurs if the answer fits in a single register – in other words, the values of the result are in the range -2G to +2G - 1.

If the product fits in a single odd register, we can convert it to packed decimal with one Convert to Decimal (**CVD**) instruction. Occasionally a multiplication will generate a double precision result that doesn't fit in a single register, so we need to know how to handle this case. For double precision results, we must convert both registers to packed decimal using two **CVD**s. If the high-order bit in the odd register is a 1, **CVD** interprets the register contents as being negative. In this case the converted value is off by a factor of 2^{32}. We correct this mistake by adding 1 (2^{32}) to the even register before issuing the **CVD** on it. We then multiply the even register contents by a factor (2^{32}) to account for the fact that the even register bits represent larger powers of two than the odd register bits. Finally, we add

the two values together arriving at a packed-decimal equivalent. I include that code below for future reference. This example converts a double precision integer in R6 and R7 to packed decimal in PRODPK.

```
            LTR   R7,R7
            BNM   CONVERT
            A     R6,=F'1'
CONVERT     EQU   *
            CVD   R6,PRODPKR
            MP    PRODPK,FACTOR
            CVD   R7,DBLWD
            AP    PRODPK,DBLWD
            ...
DBLWD       DS    D
PRODPK      DS   0PL16  LARGEST PACKED FIELD (31 DIGITS)
PRODPKL     DC    XL8'0000000000000000'
PRODPKR     DC    XL8'0000000000000000'
FACTOR      DC    P'4294967296'   2^32
```

Binary Division

Binary division also requires us to use an even/odd consecutive pair of registers. Unlike multiplication, the pair of registers must both be initialized before issuing the Divide instruction. Together, the register pair initially represents a 64-bit integer that is the dividend.

Let's examine a representative division.

```
          L     R6,X
          SRDL  R6,32   SHIFT CONTENTS OF R6 TO R7 WITH SIGN EXTENSION
          D     R6,Y
          ST    R7,Z    STORE THE QUOTIENT IN Z - IGNORE REMAINDER IN R6
          ...
X         DC    F'26'
Y         DC    F'8'
Z         DC    F
```

The before picture is the situation after the L and SRDL. The after picture is after the division.

Before: Even Register 6: (x'00000000') Odd Register 7: (x'0000001A')
After: Odd Register 6: (x'00000002') Odd Register 7: (x'00000003')

First, 26 = x'1A', is loaded into the even register and then shifted into the odd register by invoking Shift Right Double Algebraic. This is a bit-shifting operation that treats the pair of registers as a single register (bits that are shifted out of R6 land in R7). Since this is an algebraic shift right, the sign bit fills in on the left, preserving the sign of the dividend. In this case we shift 32 bits from R6 into R7, so R7 contains 26 after the shift, and R6 has the appropriate sign throughout. Before dividing, the combined register pair now contains a 64-bit version of 26. After the Divide, the quotient is left in the odd register, and the remainder is left in the even register.

Binary Comparisons

You can set the condition code with Compare (**C**), Compare Halfword (**CH**) and Compare Register (**CR**). In the case of Compare, the contents of the operand 1 register are compared against a fullword in memory. The condition code is set to indicate how the register contents compare to the contents of memory. Here is an example:

```
          L     R8,MAXITEMS
          C     R8,COUNT
          BH    STILROOM
          ...     (COUNT EQUALS OR EXCEEDS MAXITEMS HERE)
STILROOM  EQU   *
          ...
MAXITEMS  DC    F'1000'
COUNT     DS    F
```

If MAXITEMS and COUNT had been defined as a halfwords, the only differences in the code would be the use of **LH** instead of **L**, and **CH** instead of **C**.

```
           LH    R8,MAXITEMS  FILLS R8 WITH A SIGN-EXTENDED FULLWORD
           CH    R8,COUNT     COMPARES A SIGN-EXTENDED FULLWORD
           BH    STILROOM
           ...     (COUNT EQUALS OR EXCEEDS MAXITEMS HERE)
STILROOM  EQU   *
           ...
MAXITEMS   DC    H'1000'
COUNT      DS    H
```

Keep in mind that for a LH and CH instruction, the halfword is internally sign-extended, so in fact, fullwords are being compared. The contents of COUNT are unchanged by the comparison. Finally, Compare Register (**CR**) could also have been used.

```
          L     R8,MAXITEMS
          L     R9,COUNT
          CR    R8,R9
          BH    STILROOM
          ...                 (COUNT EQUALS OR EXCEEDS MAXITEMS)
STILROOM  EQU   *
          ...
MAXITEMS  DC    F'1000'
COUNT     DS    F
```

This technique has the disadvantage of requiring a second register.

A Basic Instruction Set for Binary Data Processing

The following instructions which form a working set of instructions for binary data processing. These include:

CVB	Convert to Binary
CVD	Convert to Decimal
A	Add Fullword
AH	Add Halfword
AR	Add Register
S	Subtract Fullword
SH	Subtract Halfword
SR	Subtract Register
M	Multiply Fullword
MH	Multiply Halfword
MR	Multiply Register
D	Divide Fullword
DR	Divide Register
C	Compare Fullword
CH	Compare Halfword
CR	Compare Register
L	Load Fullword
LH	Load Halfword
LA	Load Address
LR	Load Register
LTR	Load and Test Register
ST	Store Fullword
STH	Store Halfword

Op Code	R_1X_2	B_2D_2	D_2D_2

The Convert to Binary (**CVB**) instruction takes packed decimal data and converts it to 2's complement integer data in a register. Operand 1 designates a register where the result will be stored. Operand 2 represents a doubleword storage area which contains a valid 8-byte packed decimal integer.

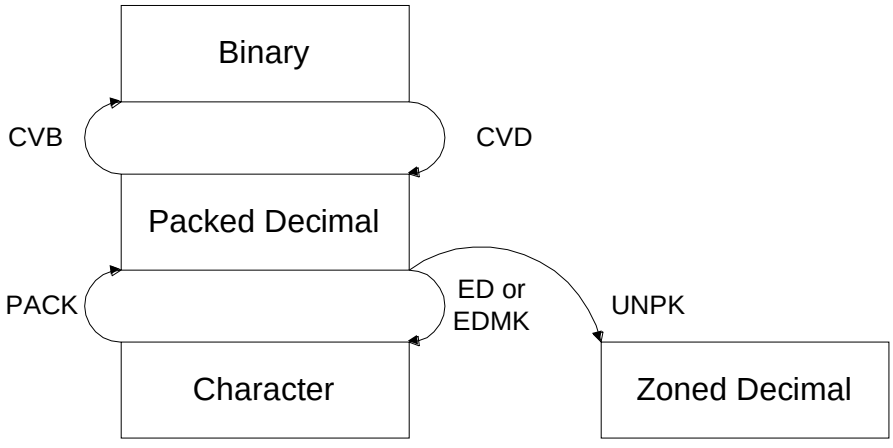
CVB can convert any packed decimal integer in the range -2,147,483,648 to +2,147,483,647. If the doubleword specified in Operand 2 contains an integer outside this range, the 32 rightmost bits of the result are placed in the Operand 1 register and a fixed-point-divide exception is recognized.

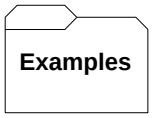
In the following example, a packed field of length 4 is converted to binary.

```
          ZAP    DOUBWORD,XPACK    STAGE THE PACKED FIELD
          CVB    R5,DOUBWORD       CHANGE TO BINARY
          ...
XPACK     DC     PL4'123'          =X'0000123C'
DOUBWORD  DS     D
```

To convert XPACK to binary, we must first move (ZAP) it to a doubleword as required by the **CVB** instruction. At the end of the conversion, R5 contains x'0000007B' = 123.

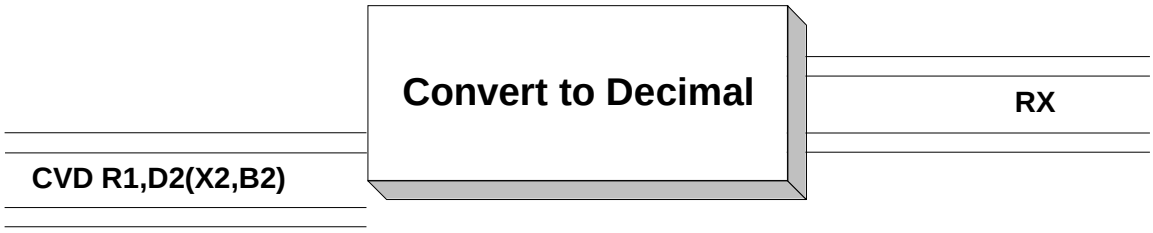
The diagram below illustrates the relationship between **CVB** and other data conversion instructions for some common data types.





Some Unrelated CVB Instructions

```
DOUBWORD DS    D
PKD1     DC    PL5'19'
PKD2     DC    P'1865'
PKD3     DC    P'-1'
         ...
         ZAP   DOUBWORD,PKD1    COPY PACKED NO TO STAGING AREA
         CVB   R8,DOUBWORD      R8 = X'00000013' = 19

         ZAP   DOUBWORD,PKD2    COPY PACKED NO TO STAGING AREA
         CVB   R8,DOUBWORD      R8 = X'00000749' = 1865

         ZAP   DOUBWORD,PKD3    COPY PACKED NO TO STAGING AREA
         CVB   R5,DOUBWORD      R5 = X'FFFFFFFF' = -1

         MVC   DOUBWORD,=C'12345678'    COPY DATA TO STAGING AREA
         CVB   R4,DOUBWORD      ABEND - DATA MUST BE PACKED

         ZAP   DOUBWORD,=P'3000000000'  COPY DATA TO STAGING AREA
         CVB   R4,DOUBWORD      ABEND - DATA > 2,147,483,647
```



Op Code	R_1X_2	B_2D_2	D_2D_2

The Convert to Decimal (**CVD**) instruction takes a 2's complement integer from a register and converts it to packed decimal data in memory. Operand 1 designates a register containing the 2's complement integer. Operand 2 represents a doubleword storage area in memory where the packed decimal data will be placed.

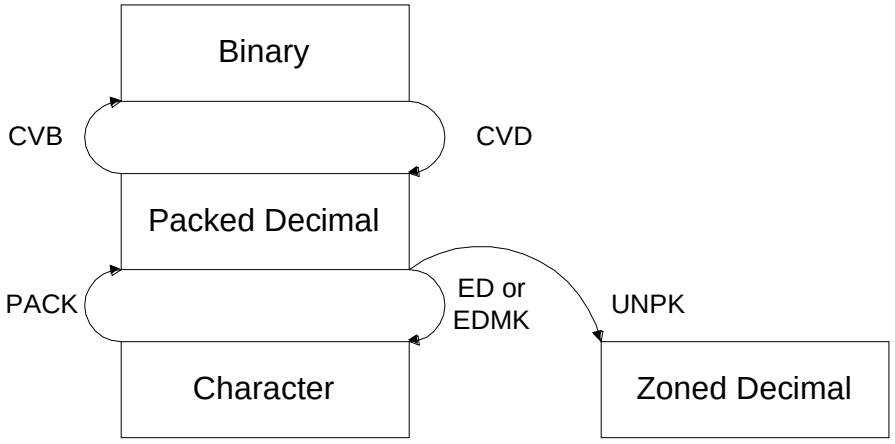
CVD can convert any 2's complement integer which is contained in a register. This includes all integers in the range -2,147,483,648 and +2,147,483,647. Since the result is placed in an 8-byte field (doubleword), no overflow can occur since there is ample room in Operand 2.

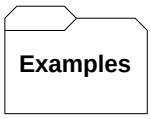
In the following example, the contents of register 5 are converted to packed decimal and placed in a doubleword. The result can be moved to a smaller field if the programmer is sure the contents will fit.

```
            CVS    R5,DOUBWORD    CHANGE IT TO PACKED DECIMAL
            ZAP    XPACK,DOUBWORD DATA WILL FIT IN 6 BYTES
            ...
XPACK       DS     PL6
DOUBWORD    DS     D
```

The integer in register 5 is converted to packed decimal and placed in a doubleword in memory. Since the doubleword contains at most 10 decimal digits (it was converted from a single register), it can be transferred to XPACK with **ZAP**.

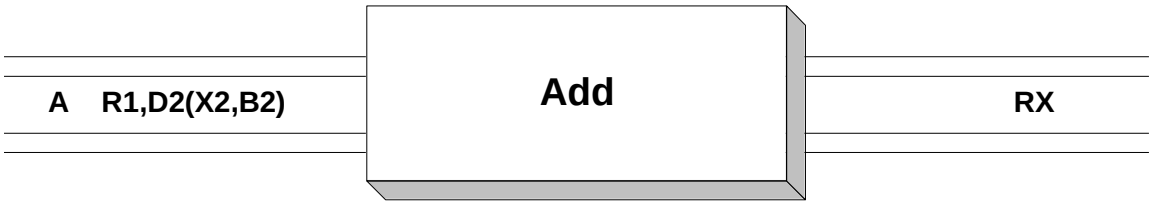
The diagram below illustrates the relationship between **CVD** and other data conversion instructions for some common data types.





Some Unrelated CVD Instructions

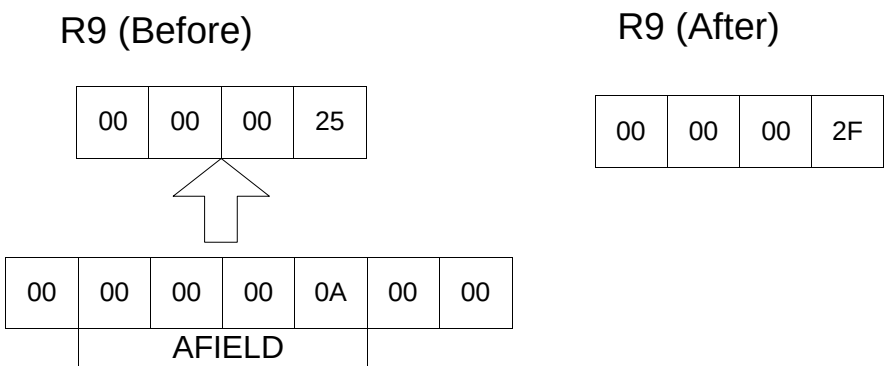
```
R7  = X'00000000' = 0
R8  = X'0000001F' = 31
R9  = X'FFFFFFFF' = -1
R10 = X'00001000' = 4096

DOUBWORD DS    D
                    ...
         CVD   R7,DOUBWORD  DOUBWORD = X'000000000000000C'
         CVD   R8,DOUBWORD  DOUBWORD = X'000000000000031C'
         CVD   R9,DOUBWORD  DOUBWORD = X'000000000000001D'
         CVD   R10,DOUBWORD DOUBWORD = X'000000000004096C'
```



Op Code	R1X2	B2D2	D2D2

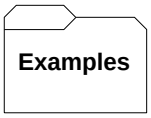
The Add (**A**) instruction performs 2's complement binary addition. Operand 1 is a register containing a fullword integer. Operand 2 specifies a fullword in memory. The fullword in memory is added to the fullword in the register and the result is stored in the register. The fullword in memory is not changed. Consider the following example,

```
A     R9,AFIELD
```



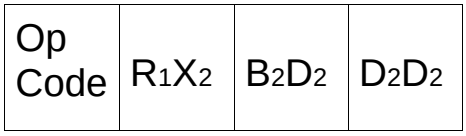
The contents of the fullword "AFIELD", x'0000000A' = 10, are added to register 9 which contains x'00000025' = 37. The sum is 47 = x'0000002F' and destroys the previous value in R9. The fullword in memory is unchanged by this operation.

Since **A** is an RX instruction, an index register may be coded as part of operand 2.



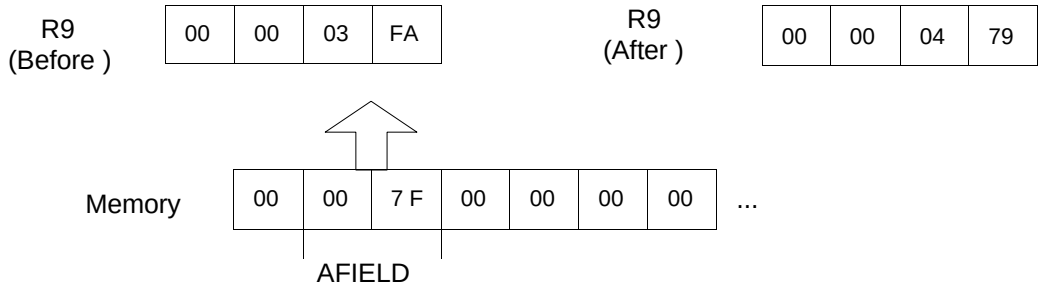
Some Unrelated Adds

```
R4 = X'FFFFFFFE' -2 IN 2'S COMPLEMENT
R5 = X'00000028' +40 IN 2'S COMPLEMENT
R6 = X'00000004' +4 IN 2'S COMPLEMENT

DOG      DC    F'4
CAT      DC    F'-4'
         ...
         A     R4,=F'20'     R4 = X'00000012' = +18
         A     R5,=F'20'     R5 = X'0000003C' = +60
         A     R6,=F'20'     R6 = X'00000018' = +24
         A     R6,=F'-5'     R6 = X'FFFFFFFF' = -1
         A     R6,CAT        R6 = X'00000000' = 0
         A     R6,DOG        R6 = X'00000008' = 8
         A     R6,DOG(R6)    R6 - X'00000000' = 0 INDEXING IS ALLOWED
```



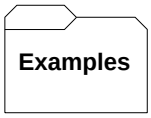
The Add Halfword (**AH**) instruction performs 2's complement binary addition. Operand 1 is a register containing a fullword integer. Operand 2 specifies a halfword in memory. The halfword in memory is sign extended (internally) and added to the fullword in the register. In other words, the halfword is converted to an arithmetically equivalent fullword before the addition. The sum is stored in the register, while the halfword in memory is not changed. Consider the following example,

```
AH R9,AFIELD
```



The contents of the halfword "AFIELD", x'007F' = 127, are added (as a fullword) to register 9 which contains x'000003FA' = 1018. The sum is 1145 = x'00000479' and destroys the previous value in R9. The halfword in memory is unchanged by this operation.

Since **AH** is an RX instruction, an index register may be coded as part of operand 2.



Some Unrelated Add Halfwords

```
R4 = X'FFFFFFF6' -10 IN 2'S COMPLEMENT
R5 = X'00000031' +49 IN 2'S COMPLEMENT
R6 = X'00000008' +8  IN 2'S COMPLEMENT

DOG DC H'4',H'9',H'-25',H'3',H'10' CONSECUTIVE HALFWORDS

         AH    R4,=H'20'   R4 = X'0000000A' = +10
         AH    R5,=H'20'   R5 = X'00000045' = +69
         AH    R6,=H'20'   R6 = X'0000001C' = +28
         AH    R6,=H'-9'   R6 = X'FFFFFFFF' = -1
         AH    R4,DOG      R6 = X'FFFFFFFA' = -6
         AH    R6,DOG      R6 = X'0000000C' = +12
         AH    R4,DOG(R6)  R6 = X'00000000' = 0 INDEXING ALLOWED
```

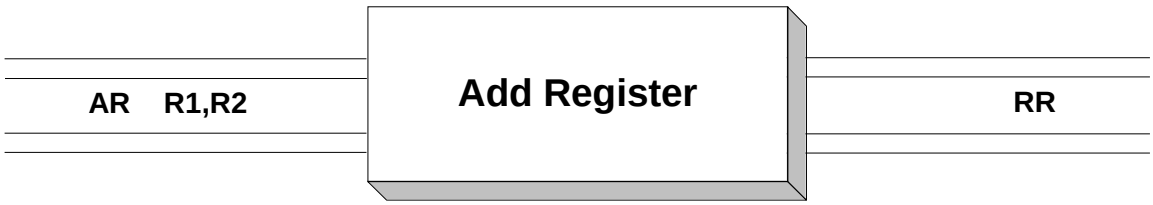
Tips

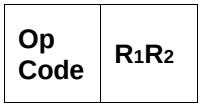
1) A common error is to code an **AH** when the second operand is not a halfword. For example:

```
AMOUNT DC F'20'         AMOUNT = X'00000014'
...
AH    R5,AMOUNT
```

The assembler will not complain about your code, but the halfword instruction will only access the first two bytes of the AMOUNT field (x'0000').

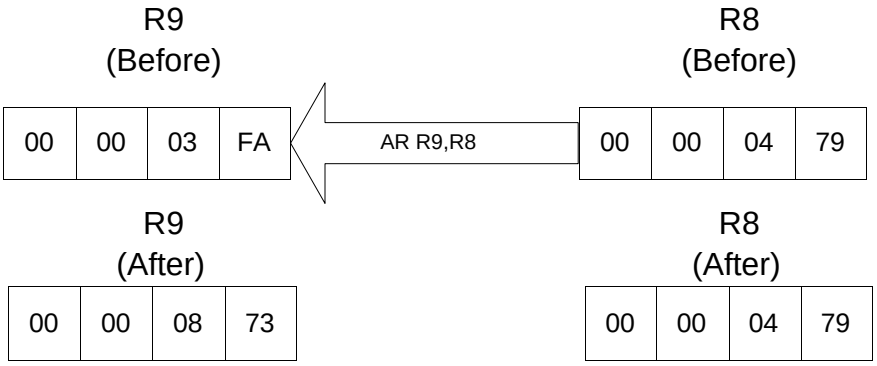
2) Give some thought as to whether to use halfword instructions and data - memory is cheap!





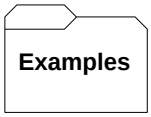
The Add Register (**AR**) instruction performs 2's complement binary addition. Operand 1 is a register containing a fullword integer. Operand 2 specifies a register as well. The fullword in Operand 2 is added to the fullword in Operand 1, and the sum replaces the contents of Operand 1. Operand 2 in unchanged by this operation except when Operand 1 and 2 refer to the same register. Consider the following example,

```
AR R9,R8
```



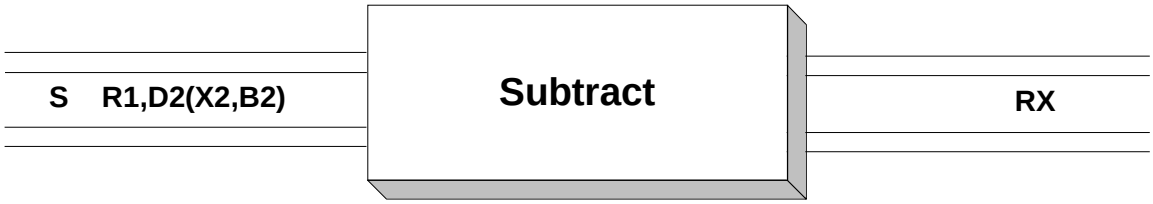
The contents of the fullword in register 8, x'00000479', are added to the contents of register 9 which contains x'000003FA'. The sum is x'00000873' and replaces the previous value in R9. The contents of register 8 are unchanged by this operation.

The condition code is set by this instruction to zero (0) if the result is zero, it is set to minus (1) if the result is negative, and to plus (2) if the result is positive.



Some Unrelated Add Register Instructions

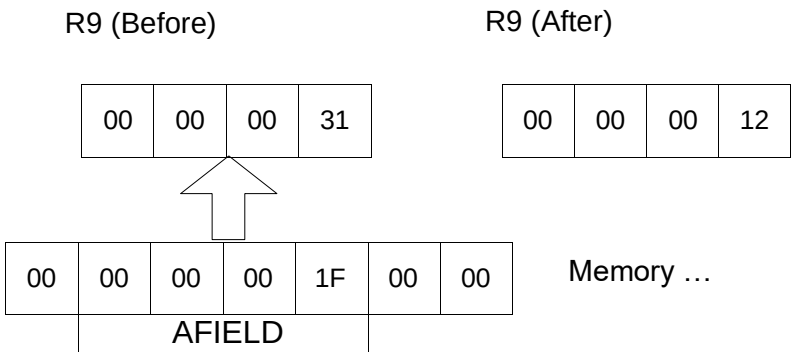
```
R4 = X'FFFFFFFE' -2  IN 2'S COMPLEMENT
R5 = X'00000028' +40 IN 2'S COMPLEMENT
R6 = X'00000004' +4  IN 2'S COMPLEMENT

         AR   R4,R5      R4 = X'00000026' = +38
         AR   R4,R4      R4 = X'FFFFFFFC' = -4
         AR   R5,R6      R5 = X'0000002C' = +44
         AR   R6,R5      R6 = X'0000002C' = +44
```



Op Code	R_1X_2	B_2D_2	D_2D_2

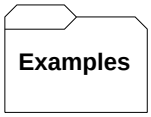
The Subtract (**S**) instruction performs 2's complement binary subtraction. Operand 1 is a register containing a fullword integer. Operand 2 specifies a fullword in memory. The fullword in memory is subtracted from the fullword in the register and the result is placed in the register. The fullword in memory is not changed. Consider the following example,

```
S     R9,AFIELD
```



The contents of the fullword AFIELD, x'0000001F' = 31, are subtracted from register 9 which contains x'00000031' = 49. The difference is 18 = x'00000012' destroying the previous value in R9. The fullword in memory is unchanged by this operation.

Since **S** is an RX instruction, an index register may be coded as part of operand 2.



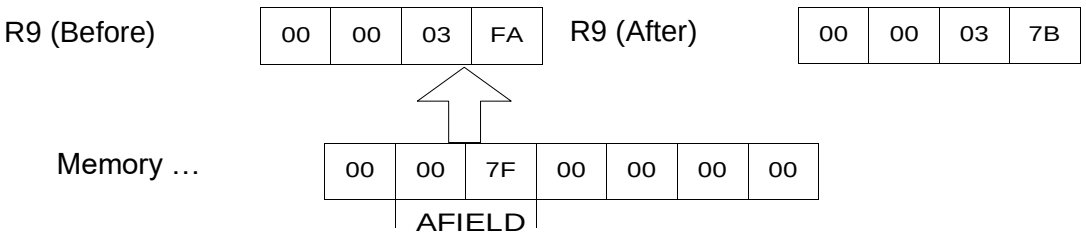
Some Unrelated Subtracts

```
R4 = X'FFFFFFD5' -43 IN 2'S COMPLEMENT
R5 = X'00000028' +40 IN 2'S COMPLEMENT
R6 = X'00000004' +4  IN 2'S COMPLEMENT

DOG      DC    F'35'
CAT      DC    F'4'
         ...
         S     R4,=F'20    R4 = X'FFFFFFC1' = -63
         S     R5,=F'-20'  R5 = X'0000003C' = +60
         S     R6,=F'20    R6 = X'FFFFFFF0' = -16
         S     R6,=F'-5'   R6 = X'00000009' = +9
         S     R6,CAT      R6 = X'00000000' = 0
         S     R5,DOG      R5 = X'00000005' = +5
         S     R6,DOG(R6)  R6 = X'00000000' = 0  INDEXING ALLOWED
```

Op Code	R_1X_2	B_2D_2	D_2D_2

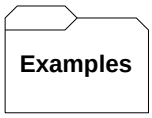
The Subtract Halfword (**SH**) instruction performs 2's complement binary subtraction. Operand 1 is a register containing a fullword integer. Operand 2 specifies a halfword in memory. The halfword in memory is sign extended (internally) and subtracted from the fullword in the register. The result remains in the register. The halfword in memory is not changed. Consider the following example,

```
SH    R9,AFIELD
```



The contents of the halfword "AFIELD", x'007F' = 127, are subtracted (as a fullword) from register 9 which contains x'000003FA' = 1018. The difference, 891 = x'0000037B', destroys the previous value in R9. The halfword in memory is unchanged by this operation.

Since **SH** is an RX instruction, an index register may be coded as part of operand 2.



Some Unrelated Subtract Halfword Instructions

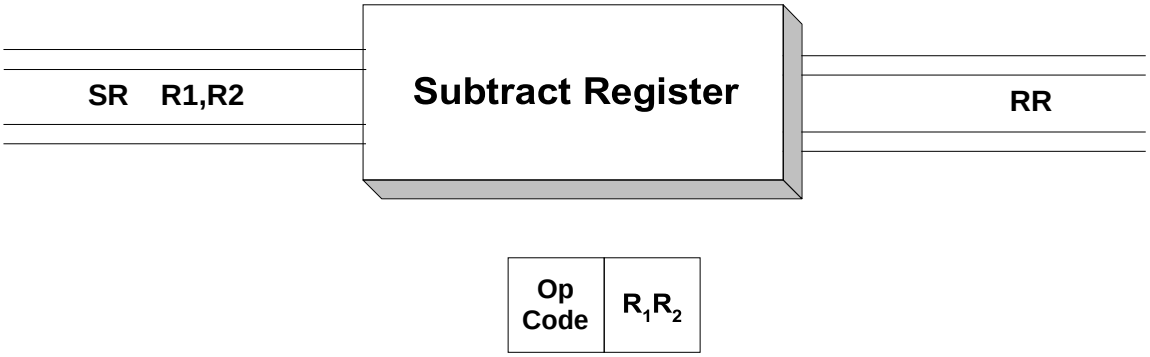
```
R4 = X'FFFFFFF0'   -16 IN 2'S COMPLEMENT
R5 = X'00000025'   +37 IN 2'S COMPLEMENT
R6 = X'00000004'    +4 IN 2'S COMPLEMENT

DOG       DC    H'4',H'9',H'37',H'3'   CONSECUTIVE HALFWORDS
          SH    R4,=H'20       R4 = X'FFFFFFDC' = -36
          SH    R5,=H'20'      R5 = X'00000011' = +17
          SH    R6,=H'4'       R6 = X'00000000' = 0
          SH    R4,=H'-9'      R4 = X'FFFFFFF9' = -7
          SH    R5,=DOG        R5 = X'00000021' = +33
          SH    R5,DOG(R6)     R5 = X'00000000' = 0  INDEXING ALLOWED
```

Tips

A common error is to code an **SH** when the second operand is not a halfword. For example:

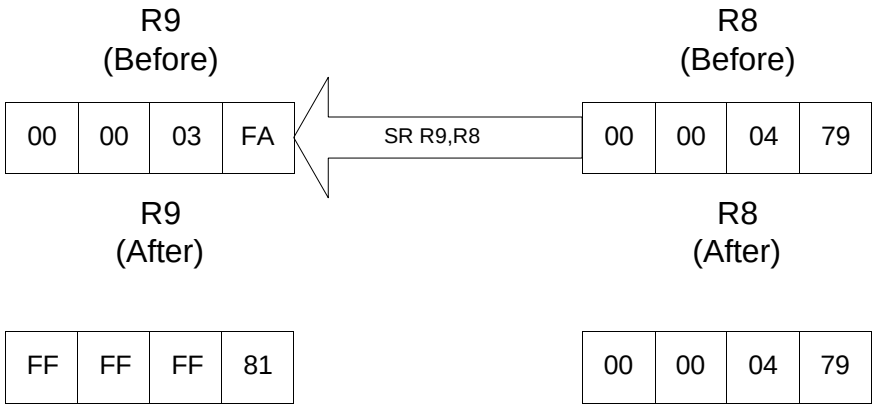
```
AMOUNT    DC    F'20'   AMOUNT = X'00000014'
          ...
          SH    R5,AMOUNT
```

The assembler will not complain about your code, but the halfword instruction will only access the first two bytes of the AMOUNT field (x'0000').



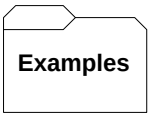
The Subtract Register (**SR**) instruction performs 2's complement binary subtraction. Operand 1 is a register containing a fullword integer. Operand 2 specifies a register as well. The fullword in Operand 2 is subtracted from the fullword in Operand 1, and the difference replaces the contents of Operand 1. Operand 2 in unchanged by this operation except when Operands 1 and 2 refer to the same register. Consider the following example,

```
SR     R9,R8
```



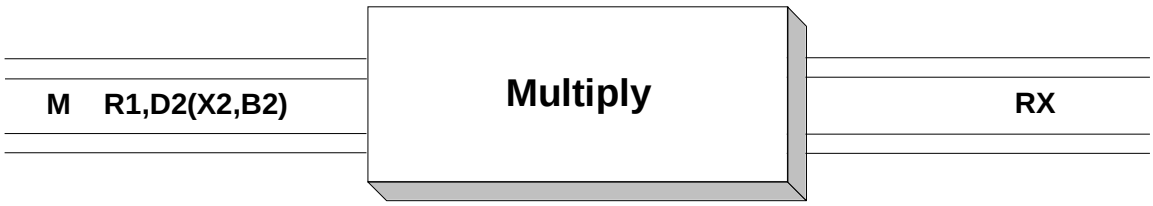
The contents of the fullword in register 8, x'00000479', are subtracted from the contents of register 9 which contains x'000003FA' . The difference is x'FFFFF181' and replaces the previous value in R9. The contents of register 8 are unchanged by this operation.

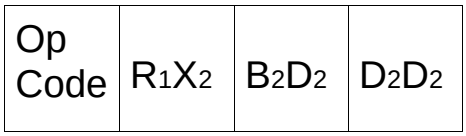
The condition code is set by this instruction to zero if the result is zero, minus if the result is negative, and plus if the result is positive.



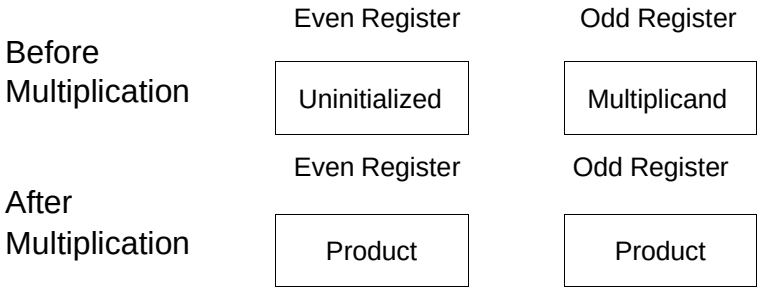
Some Unrelated Subtract Register Instructions

```
R4 = X'FFFFFFFE'  -2 IN 2'S COMPLEMENT
R5 = X'00000028' +40 IN 2'S COMPLEMENT
R6 = X'00000004' +4 IN 2'S COMPLEMENT

       SR    R4,R4     R4 = X'00000000'  =   0
       SR    R5,R4     R5 = X'0000002A'  = +42
       SR    R5,R6     R5 = X'00000024'  = +36
       SR    R6,R5     R6 = X'FFFFFFDC'  = -36
```





The Multiply (**M**) instruction performs 2's complement binary multiplication. Operand 1 names an even register of an "even-odd" consecutive register pair. For instance, R2 would be used to name the R2 / R3 even-odd register pair, and R8 would be used to name the R8 / R9 even-odd register pair. Operand 2 is the name of a fullword in memory containing the multiplier. Before the multiplication, the even register can be left uninitialized, while the odd register contains the multiplicand. After the multiplication, the product occupies the even-odd register pair in 2's complement format.

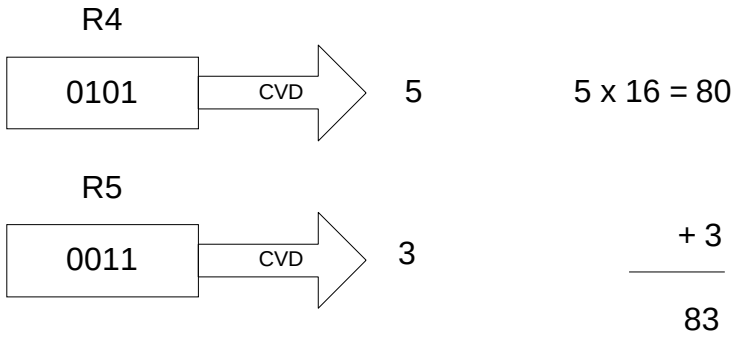


If the product is less than $2^{31} - 1 = 2{,}147{,}483{,}647$ then the answer can be found in the odd register. We may then use **CVD** and **ED** to print the product. Otherwise, the even-odd pair must be treated as a large (64 bit) 2's complement integer. First, let's look at an example multiplication.

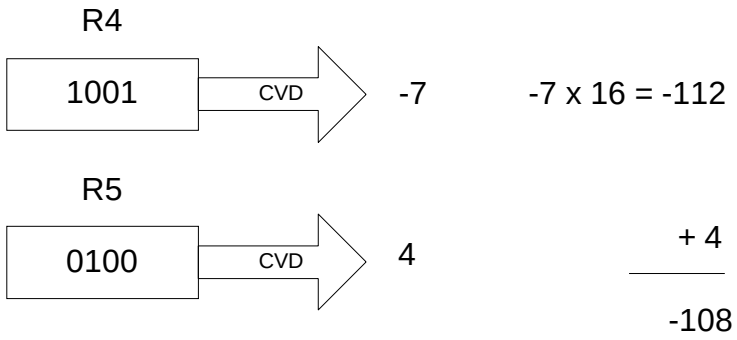
```
AWORD     DC    F'47'
TEN       DC    F'10'
          ...
          L     R9,AWORD     PUT MULTIPLICAND IN ODD REGISTER
          M     R8,TEN       MULTIPLY 47 BY 10
```

First the multiplicand, 47, is loaded into the odd register. The even register is left uninitialized. The multiplier, AWORD, contains a 10 and is not affected by the multiplication. After the multiplication, the register pair R8 / R9 contains a 64 bit 2's complement integer. Since the product is sufficiently small, R9 by itself contains a valid representation of the product.

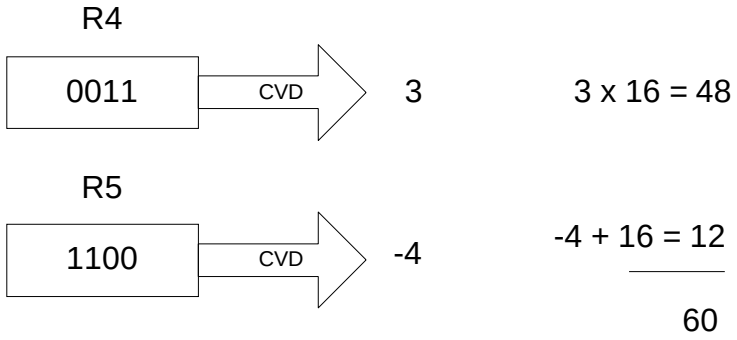
When the product will not fit in the odd register, we must provide special handling to convert the product to a packed representation. If it takes two registers to hold the result, we will call the answer a "double precision" result. Unfortunately, there is no single instruction that will convert a 2's complement double precision integer to a packed decimal format. **CVD** can be used to convert a single register to packed format, so we will investigate again how this instruction can be used on both registers. To simplify the computations, we will assume that the registers contain 4 bits instead of 32. Suppose a multiplication has produced a double precision product of 83 in registers R4 and R5. Then, since 83 = B'01010011', and assuming 4-bit registers, R4 = B'0101' and R5 = B'0011'. If we use **CVD** to convert R4 we would get 5, when in fact, the bits in R4 represent 80 if we look at the 2's complement integer contained in R4 and R5. We are off by a factor of 2^4 =16 since 5 x 16 = 80. If we use **CVD** to convert R5 we would get 3, which is what the bits in the double precision integer represent. The true answer can be recovered by adding (5 x16) + 3. This is illustrated in the diagram below.



This procedure also works for some negative double precision integers. Consider the double precision integer -108. Using 4-bit registers R4 and R5, we see that R4 contains B'1001' and R5 contains B'0100'. **CVD** converts R4 to -7 when, in fact, the bits in R4, B'1001', represent -112 = -7 x 16. R5 = B '0100' is converted to 4. Adding -112 + 4 we get the correct double precision answer -108. This is illustrated below.



A problem with this method occurs when the odd register contains a 1 in the high-order bit. Consider the double precision integer 60 = B'00111100'. Assume R4 contains B'0011' and R5 contains B'1100'. The conversion is illustrated below.



R4 is converted to 3, multiplied by 16, and correctly converted to 48. On the other hand, R5 is converted to -4 since the high order bit was a 1. R5 should have been converted to 12. We are off by a factor or 16. If we add 16, the conversion to 12 will be correct.

These examples lead us to a conversion algorithm for double precision results:

1) Test the odd register to see if it is negative. If it is, we need to add 2^{32} = 4,294,967,296 (we are using 32-bit registers instead of 4-bit registers) to the final result.

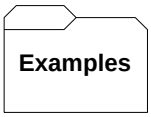
An easy way to do this is by adding 1 to the even register - the rightmost bit in the even register represents 2^{32}.

2) Convert the even register to packed decimal and multiply the result by 2^{32}.

3) Add in the result of converting the odd register to packed decimal.

Here is an example in assembler language of the algorithm described above. The example illustrates how a double precision result in R4 and R5 could be converted to packed decimal in a doubleword.

```
            LTR    R5,R5            DOES R5 LOOK NEGATIVE?
            BNM    LOOKSPOS         DON'T ADD TO R4 IF POSITIVE
            A      R4,=F'1'         WE ARE OFF BY 2 TO THE 32 POWER
LOOKSPOS    EQU    *
            CVD    R4,RESULTRT      CONVERT AND GET READY..
            MP     RESULT,TWOTO32   MULTIPLY BY 2 TO THE 32
            CVD    R5,DOUBWORD      CONVERT ODD REG TO DECIMAL
            AP     RESULT,DOUBWORD ADD THE TWO COMPONENTS
            ...
RESULT      DS    0PL16   NEED A LARGE AREA TO HOLD DOUBLE PRECISION
RESULTLF    DC     X'00000000' NEEDS TO BE 0'S AFTER CONVERTING R4
RESULTRT    DS     PL6             WORK AREA FOR R4
TWOTO32     DC     P'4294967296'   2 TO THE 32ND POWER
DOUBWORD    DS     D               CONVERSION AREA FOR CVD
```

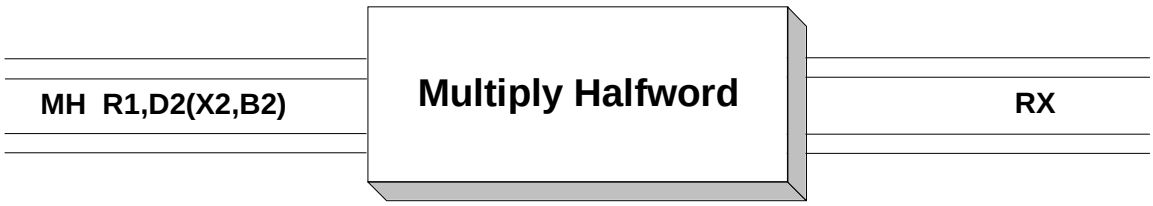


Some Unrelated Multiply Instructions

```
L    R7,=F'100'   MULTIPLICAND GOES IN THE ODD REGISTER
M    R6,=F'10     R6 = X'00000000' = 0, R7 = X'000003E8' = 1000

L    R7,=F'3'     MULTIPLICAND GOES IN THE ODD REGISTER
M    R6,=F'-2'    R6 = X'FFFFFFFF', R7 = X'FFFFFFFA' = -6

L    R3,=F'8'     MULTIPLICAND GOES IN THE ODD REGISTER
M    R2,=F'1'     R2 = X'00000000', R3 = X'00000008'

L    R3,=F'8'     MULTIPLICAND GOES IN THE ODD REGISTER
M    R2,=F'0'     R2 = X'00000000', R3 = X'00000000'

L    R5,=X'FFFFFFFF'    ALL 1'S IN ODD REG
M    R4,=F'2'     MULTIPLYING BY 2 SHIFTS ALL BITS 1 BIT LEFT
                  R4 = X'00000001', R5 = X'FFFFFFFE', THIS IS A
                  DOUBLE PRECISION RESULT
```

Tips

1) Know your data! In most cases, the product of a multiplication will fit in the odd register where it can easily be converted back to packed decimal. If you have any doubts about the size of a generated product, you must convert the double precision result from both the even and odd registers as described above.

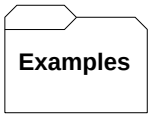


Op Code	R_1X_2	B_2D_2	D_2D_2

The Multiply Halfword (**MH**) instruction performs 2's complement binary multiplication. Operand 1 names a single register (even or odd) which will contain the multiplicand. Operand 2 is the name of a halfword in memory containing the multiplier. After the multiplication, the product is left in Operand 1, destroying the multiplicand. It is possible to generate a product that will not fit in a single register, but an overflow will not be indicated. Leftmost bits in the product could be truncated to 32 bits in Operand 1. The programmer must be aware of the limits of the data being processed and protect against the possibilities of overflows. Select a fullword multiplication (**M**) if you are unsure if your data will overflow a single register.

As an example, assume you want to multiply a fullword field called COST by a halfword field called NOITEMS. The following code would accomplish this task and leave the product in register 5.

```
L       R5,COST
MH      R5,NOITEMS
```

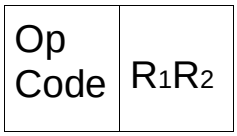


Some Unrelated Multiply Halfword Instructions

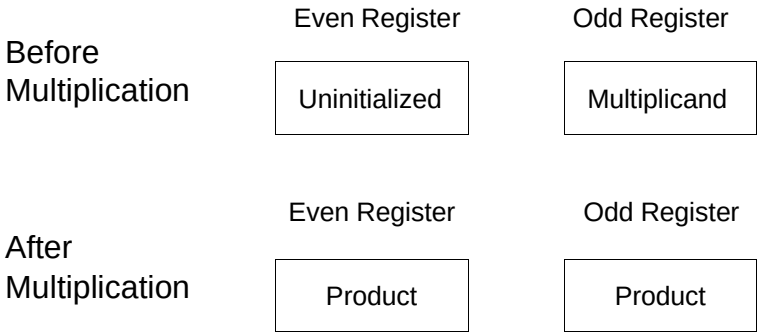
```
L      R6,=F'100'       MULTIPLICAND GOES IN ANY REGISTER
MH     R6,=F'10         R6 = X'000003E8' = 1000

L      R7,=F'3'         MULTIPLICAND GOES IN ANY REGISTER
MH     R7,=F'-2'        R7 = X'FFFFFFFA' = -6

L      R3,=F'8'         MULTIPLICAND GOES IN ANY REGISTER
MH     R3,=F'2'         R3 = X'00000010' = 16

L      R3,=F'8'         MULTIPLICAND GOES IN ANY REGISTER
MH     R3,=F'0'         R3 = X'00000000' = 0

L      R4,=X'7FFFFFFF' LARGEST POSITIVE NUMBER IN A SINGLE REG.
MH     R4,=F'2'         MULTIPLYING BY 2 CAUSES OVERFLOW
                        R4 = X'FFFFFFFE' = -2 INCORRECT RESULT
```

Tips

1) Know your data! To use **MH**, one of the operands must be small enough to fit in a halfword. This range is -32768 to +32,767. The product of the multiplication must fit in a single register where the range of integers is -2,147,483,648 to +2,147,483,647. In many cases, the product of a multiplication will fit in a single register. If you have any doubts about the size of a generated product, use **M** instead of **MH**.



The Multiply Register (**MR**) instruction performs 2's complement binary multiplication. Operand 1 names an even register of an "even-odd" consecutive register pair. For instance, R2 would be used to name the R2 / R3 even-odd register pair, and R8 would be used to name the R8 / R9 even-odd register pair. Operand 2 names a register containing the multiplier. Before the multiplication, the even register can be left uninitialized, while the odd register contains the multiplicand. After the multiplication, the product occupies the even-odd register pair in 2's complement format.



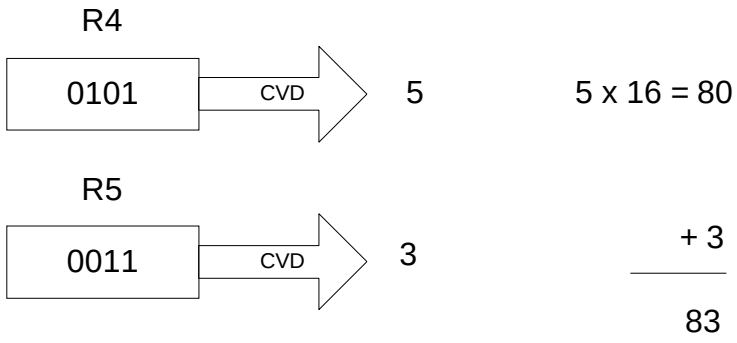
If the product is less than $2^{31} - 1 = 2,147,483,647$ then the answer can be found in the odd register. We may then use **CVD** and **ED** to print the product. Otherwise, the even-odd pair must be treated as a large (64 bit) 2's complement integer. First, let's look at an example multiplication.

```
L       R9,=F'47'      PUT MULTIPLICAND IN ODD REGISTER
L       R5,=F'10'      PUT MULIPLIER IN A REGISTER
MR      R8,R5          MULTIPLY EVEN/ODD PAIR(8/9)TIMES R5
```

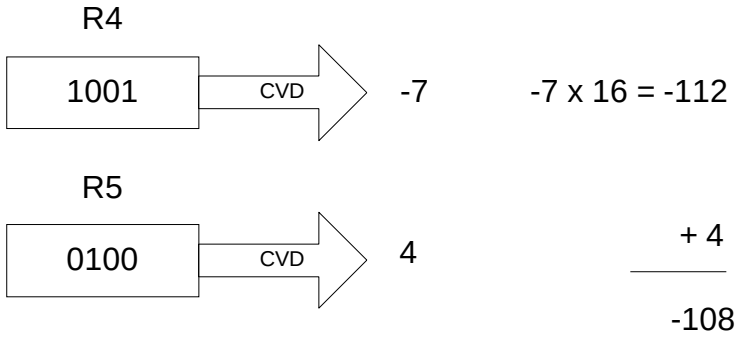
The multiplicand, 47, is loaded into the odd register. The even register is left uninitialized and is shown above to contain x'000AB356' (this is arbitrary). The integer 10 is loaded into R5 and acts as the multiplier. After the multiplication, the register pair R8 / R9 contains a 64-bit 2's complement integer representing the product. Since the product is sufficiently small, R9 contains a valid representation of the product.

When the product will not fit in the odd register, we must provide special handling to convert the product to a packed representation. If it takes two registers to hold the result, we will call the answer a "double precision" result. Unfortunately, there is no single instruction that will convert a 2's complement double precision integer to a packed decimal format. **CVD** can be used to convert a single register to packed format, so we will investigate how this instruction can be used on both registers. For the purposes of demonstration, and to simplify the computations, we will assume that the registers contain 4 bits instead of 32. Suppose a multiplication has produced a double precision product of 83 in registers R4 and R5.

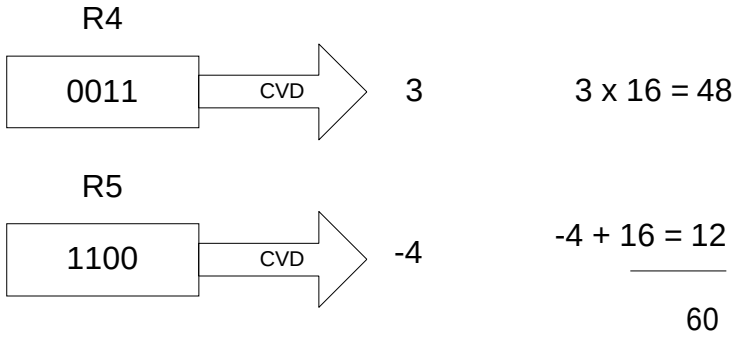
Since 83 = B'01010011', and assuming 4-bit registers to make things simpler, R4 = B'0101' and R5= B'0011'. If we use **CVD** to convert R4 we get a 5, when in fact, the bits in R4 represent 80 as part of the R4/R5 pair. We are off by a factor of 2^4 =16 since 5 x 16 = 80. If we use **CVD** to convert R5 we would get 3, which is correct. The true answer can be recovered by adding (5 x16) + 3. This is illustrated in the diagram below.



This procedure also works for some negative double precision integers. Consider the double precision integer -108. Using 4-bit registers R4 and R5, we see that R4 contains B'1001' and R5 contains B'0100'. **CVD** converts R4 to -7 when, in fact, the bits in R4, B'1001', represent -112 = -7 x 16. R5 = B '0100' is converted to 4. Adding -112 + 4 we get the correct double precision answer -108. This is illustrated below.



A problem with this method occurs when the odd register contains a 1 in the high-order bit. Consider the double precision integer 60 = B'00111100'. Assume R4 contains B'0011' and R5 contains B'1100'. The conversion is illustrated below.



R4 is converted to 3, multiplied by 16, and correctly converted to 48. On the other hand, R5 is converted to -4 since the high order bit was a 1. R5 should have been converted to 12. We are off by a factor or 16. If we add 16, the conversion to 12 will be correct.

These examples lead us to a conversion algorithm for double precision results:

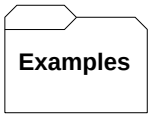
1) Test the odd register to see if it is negative. If it is, we need to add 2^{32} = 4,294,967,296 (we are using 32-bit registers instead of 4-bit registers) to the final result. An easy way to do this is by adding 1 to the even register - the rightmost bit in the even register represents 2^{32}.

2) Convert the even register to packed decimal and multiply the result by 2^{32}.

3) Add in the result of converting the odd register to packed decimal.

Here is an example in assembler language of the algorithm described above. The example illustrates how a double precision result in R4 and R5 could be converted to packed decimal in a doubleword.

```
            LTR R5,R5           DOES R5 LOOK NEGATIVE?
            BNM LOOKSPOS        DON'T ADD TO R4 IF POSITIVE
            A R4,=F'1'          WE ARE OFF BY 2 TO THE 32 POWER
LOOKSPOS    EQU *
            CVD R4,RESULTRT     CONVERT AND GET READY..
            MP RESULT,TWOTO32   MULTIPLY BY 2 TO THE 32
            CVD R5,DOUBWORD     CONVERT ODD REG TO DECIMAL
            AP RESULT,DOUBWORD  ADD THE TWO COMPONENTS
            ...
RESULT      DS 0PL16           NEED A LARGE AREA TO HOLD DOUBLE PRECISION
RESULTLF    DC X'00000000'     NEEDS TO BE 0'S AFTER CONVERTING R4
RESULTRT    DS PL6             WORK AREA FOR R4
TWOTO32     DC P'4294967296'   2 TO THE 32ND POWER
DOUBWORD    DS D               CONVERSION AREA FOR CVD
```



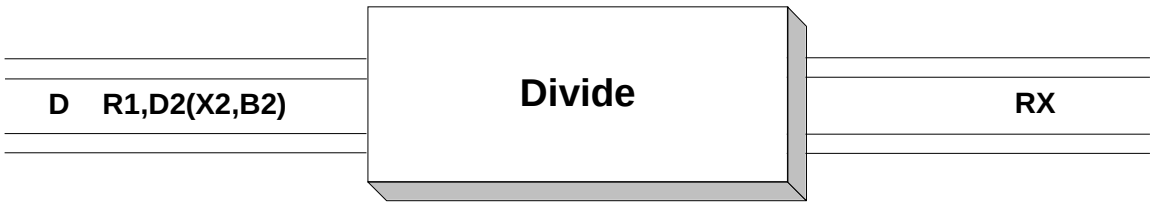
Some Unrelated Multiply Register Instructions

```
L     R7,=F'100'       MULTIPLICAND GOES IN THE ODD REGISTER
L     R9,=F'10'        MULTIPLIER GOES IN R9
MR    R6,R9           R6 = X'00000000' = 0, R7 = X'000003E8' = 1000

L     R7,=F'3'         MULTIPLICAND GOES IN THE ODD REGISTER
L     R6,=F'-2'        MULTIPLIER CAN OCCUPY THE EVEN REGISTER!
MR    R6,R6            OPERAND 1 INDICATES EVEN/ODD PAIR R6/R7
                       OPERAND 2 REFERENCES THE MULTIPLIER - R6
                       AFTERWARD:  R7 = X'FFFFFFFA'
                                   R6 = X'FFFFFFFF'

L     R3,=F'8'         MULTIPLICAND GOES IN THE ODD REGISTER
L     R2,=F'1'         MULTIPLIER GOES IN R2
MR    R2,R2            R2 = X'00000000', R3 = X'00000008'

L     R3,=F'8'         MULTIPLICAND GOES IN THE ODD REGISTER
L     R8,=F'0'         MULTIPLIER GOES IN R8
MR    R2,R8            R2 = X'00000000', R3 = X'00000000'

L     R5,=X'FFFFFFFF'  ALL 1'S IN ODD REG = -1
L     R9,=F'2'         MULTIPLIER GOES IN R9
MR    R4,R9
                       R4 = X'FFFFFFFF', R5 = X'FFFFFFFE',
                       THIS IS A DOUBLE PRECISION RESULT = -2
```

Tips

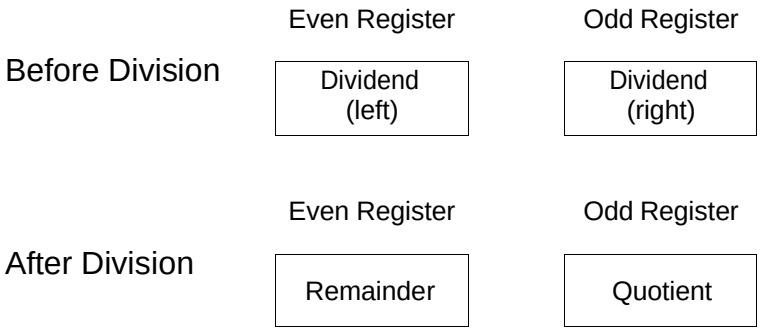
1) Know your data! In most cases, the product of a multiplication will fit in the odd register where it can easily be converted back to packed decimal. If you have any doubts about the size of a generated product, you must convert the double precision result from both the even and odd registers as described above.

2) Rather than leave the even register uninitialized, you can use it to hold the multiplier. If you do this, the multiplier will be destroyed since the product occupies the even/odd register pair.



Op Code	R_1X_2	B_2D_2	D_2D_2

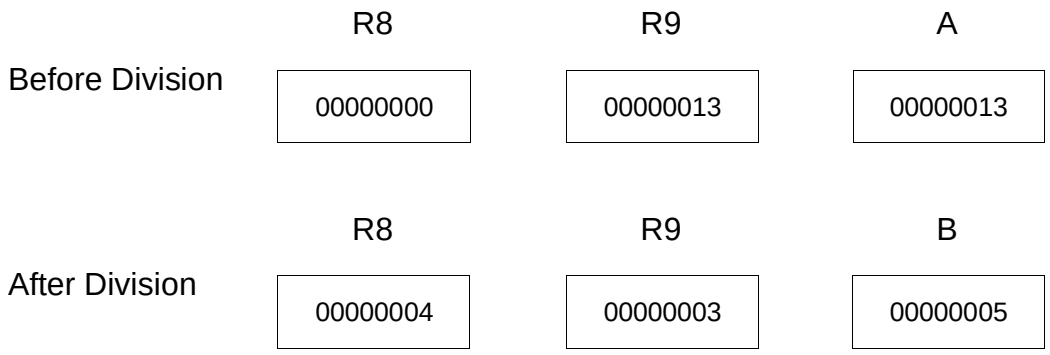
The Divide (**D**) instruction performs 2's complement binary integer division and returns a quotient and a remainder. Operand 1 names an even register of an "even-odd" consecutive register pair. For instance, R2 would be used to name the R2 / R3 even-odd register pair, and R8 would be used to name the R8 / R9 even-odd register pair. Operand 2 is the name of a fullword in memory containing the divisor. Before the division, the even-odd register pair must be initialized with the dividend, which is effectively a 64-bit 2's complement integer. After the division, the remainder is contained in the even register and the quotient is contained in the odd register. The sign of the quotient is determined by the rules of algebra. The sign of the remainder will be the same as the dividend.



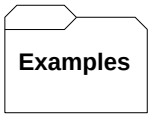
In preparing to divide a fullword, a common practice is to load the fullword in the even register and algebraically shift it to the odd register. This practice propagates the appropriate sign bit throughout the even register and successfully initializes the even/odd register pair. Here is an example where we compute A/B where A and B are fullwords.

```
            L       R8,A        PUT THE DIVIDEND IN THE EVEN REGISTER
            SRDA    R8,32       ALGEBRAICALLY SHIFT R8 INTO R9
            D       R8,B        DIVIDE A BY B
            ...
A           DC      F'19'       DIVIDEND
B           DC      F'5'        DIVISOR
```

The diagram below illustrates the above division just after R8 has been shifted.



The diagram illustrates that the result of the integer division of 19 by 5 is a remainder of 4 in R8, and a quotient of 3 in R9.



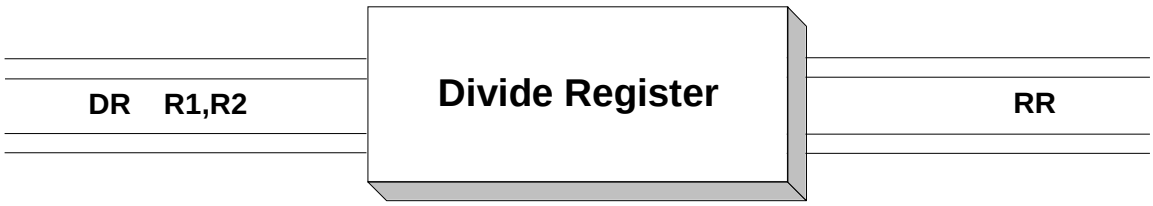
Some Unrelated Divide Instructions

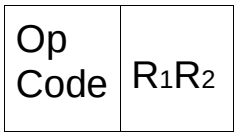
```
L       R6,=F'100'  DIVIDEND INITIALLY GOES IN THE EVEN REGISTER
SRDA    R6,32       ... AND IS SHIFTED TO THE ODD REGISTER
D       R6,=F'10'   ... BEFORE DIVIDING
                    R6 (REMAINDER) = X'00000000',
                    R7 (QUOTIENT) = X'0000000A'

L       R6,=F'100'  DIVIDEND INITIALLY GOES IN THE EVEN REGISTER
SRDA    R6,32       ... AND IS SHIFTED TO THE ODD REGISTER
D       R6,=F'8'    ... BEFORE DIVIDING
                    R6 (REMAINDER) = X'00000004',
                    R7 (QUOTIENT) = X'0000000C'

L       R6,=F'100'  DIVIDEND INITIALLY GOES IN THE EVEN REGISTER
SRDA    R6,32       ... AND IS SHIFTED TO THE ODD REGISTER
D       R6,=F'0'    ... BEFORE DIVIDING
                    ABEND - DIVISION BY 0 NOT ALLOWED

L       R6,=F'-100' DIVIDEND INITIALLY GOES IN THE EVEN REGISTER
SRDA    R6,32       ... AND IS SHIFTED TO THE ODD REGISTER
D       R6,=F'-8'   ... BEFORE DIVIDING
                    R6 (REMAINDER) =X'FFFFFFFC' = -4
                    R7 (QUOTIENT) = X'0000000C'
```

Tips

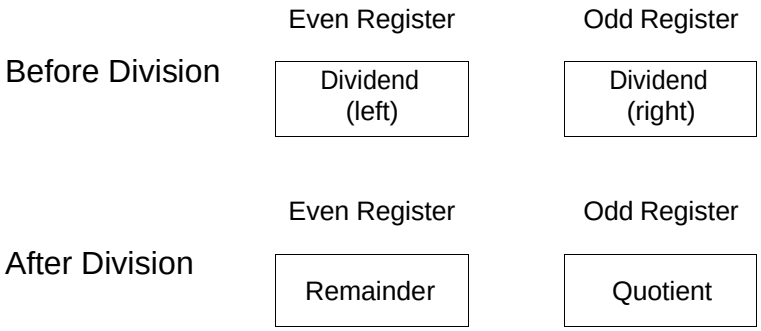
1. Know your data! If the divisor might be zero, you must protect your divisions by testing the divisor beforehand.

```
        CLC  DIVISOR,=F'0     IS THE DIVISOR 0?
                              THIS ASSUMES DIVISOR IS A FULLWORD
        BE   ZERODIV          BRANCH IF DIVISOR IS 0
        D    R8,DIVISOR       O.K. TO DIVIDE NOW
        ...
ZERODIV EQU *
        (CODE TO HANDLE A ZERO DIVISOR)
```





The Divide Register (**DR**) instruction performs 2's complement binary integer division and returns a quotient and a remainder. Operand 1 names an even register of an "even-odd" consecutive register pair. For instance, R2 would be used to name the R2 / R3 even-odd register pair, and R8 would be used to name the R8 / R9 even-odd register pair. Operand 2 names a register that contains the divisor. Before the division, the even-odd register pair must be initialized with the dividend, which is effectively a 64-bit 2's complement integer. After the division, the remainder is contained in the even register and the quotient is contained in the odd register. The sign of the quotient is determined by the laws of algebra, and the sign of the remainder is the same as the sign of the dividend.

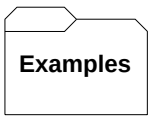


In preparing to divide a fullword, a common practice is to load the fullword in the even register and algebraically shift it to the odd register using **SRDA**. This practice propagates the appropriate sign bit throughout the even register. (If the dividend is positive, the even register is populated with binary 0's. If the dividend is negative, the even register is populated with binary 1's.) Here is an example where we compute A/B where A and B are fullwords.

```
            L       R8,A     PUT THE DIVIDEND IN THE EVEN REGISTER
            SRDA    R8,32    ALGEBRAICALLY SHIFT R8 INTO R9
            L       R5,B     PUT THE DIVISOR IN A REGISTER
            DR      R8,R5    DIVIDE A BY B
            ...
 A          DC      F'19'    DIVIDEND
 B          DC      F'5'     DIVISOR
```

The diagram below illustrates the above division just after R8 has been shifted.

The diagram illustrates that the results of the integer division of 19 by 5 is a remainder of 4 in R8, and a quotient of 3 in R9.



Some Unrelated Divide Register Instructions

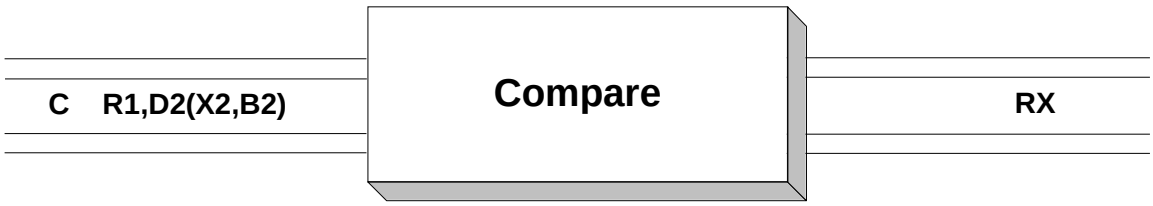
```
L        R6,=F'100'      DIVIDEND INITIALLY GOES IN THE EVEN REGISTER
SRDA     R6,32           ... AND IS SHIFTED TO THE ODD REGISTER
L        R9,=F'10'       DIVISOR GOES IN R9
DR       R6,R9           ... BEFORE DIVIDING
                         R6 (REMAINDER) = X'00000000',
                         R7 (QUOTIENT) = X'0000000A'

L        R6,=F'100'      DIVIDEND INITIALLY GOES IN THE EVEN REGISTER
SRDA     R6,32           ... AND IS SHIFTED TO THE ODD REGISTER
L        R4,=F'8'        DIVISOR GOES IN R4
DR       R6,R4           ... BEFORE DIVIDING
                         R6 (REMAINDER) = X'00000004',
                         R7 (QUOTIENT) = X'0000000C'

L        R6,=F'100'      DIVIDEND INITIALLY GOES IN THE EVEN REGISTER
SRDA     R6,32           ... AND IS SHIFTED TO THE ODD REGISTER
SR       R5,R5           ZERO OUT R5
DR       R6,R5           ... BEFORE DIVIDING ABEND
                           - DIVISION BY 0 NOT ALLOWED

L        R6,=F'-100'     DIVIDEND INITIALLY GOES IN THE EVEN REGISTER
SRDA     R6,32           ... AND IS SHIFTED TO THE ODD REGISTER
L        R10,=F'-8'      DIVISOR GOES IN R8
DR       R6,R10          ... BEFORE DIVIDING
                         R6 (REMAINDER) = X'FFFFFFFC',
                         R7 (QUOTIENT) = X'0000000C'
```

Tips

1) Know your data! If the divisor might be zero, you must protect divisions by testing the divisor beforehand.

```
            LTR   R5,R5   ASSUME DIVISOR IS IN R5
            BZ    ZERODIV   BRANCH IF DIVISOR IS 0
            DR    R8,R5     OK TO DIVIDE NOW
            ...
ZERODIV     EQU *
              (CODE TO HANDLE A ZERO DIVISOR)
```

2) Unlike the **MR** instruction, you must initialize the even register. The even/odd register pair must contain a 64-bit binary integer before you execute the instruction.

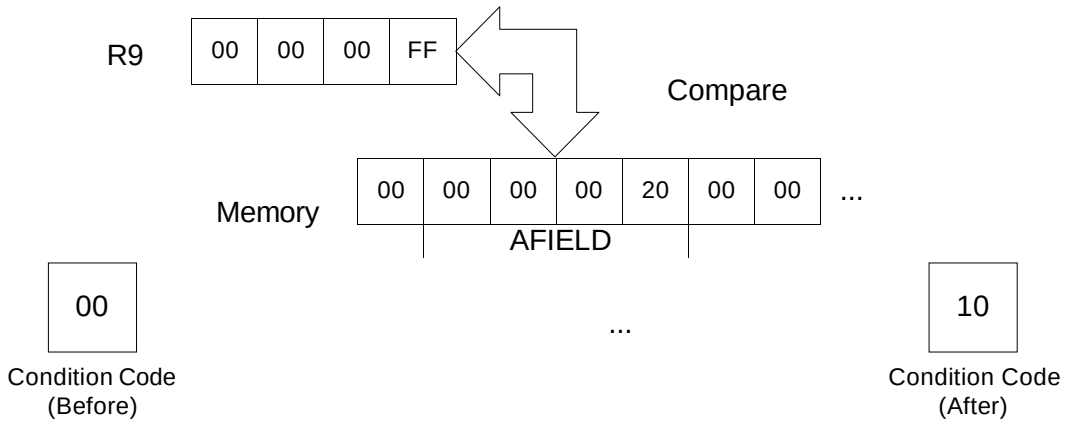


Op Code	R_1X_2	B_2D_2	D_2D_2

The Compare (**C**) instruction is used to compare a binary fullword in a register, Operand 1, with a fullword in memory, Operand 2. The operands are compared as 32-bit signed binary integers. The instruction sets the condition code to indicate how Operand 1 compares to Operand 2:

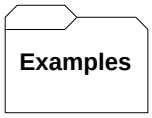
Condition Code	Meaning	Test With
0	Operand 1 = Operand 2	BE, BZ
1	Operand 1 < Operand 2	BL, BM
2	Operand 1 > Operand 2	BH, BP

```
C   R9,AFIELD
```



The contents of the fullword "AFIELD", x'00000020', is compared to the contents of register 9 which contains x'000000FF'. Since the contents of the register (Operand 1) is greater than the value than the fullword in memory (Operand 2), the condition code is set to "High". The condition code in the diagram above is specified using 2 binary digits. After comparison, the condition code is set to a binary 10 which is 2 in decimal - a "High" condition.

Since **C** is an RX instruction, an index register may be coded as part of operand 2.



Some Unrelated Compare Instructions

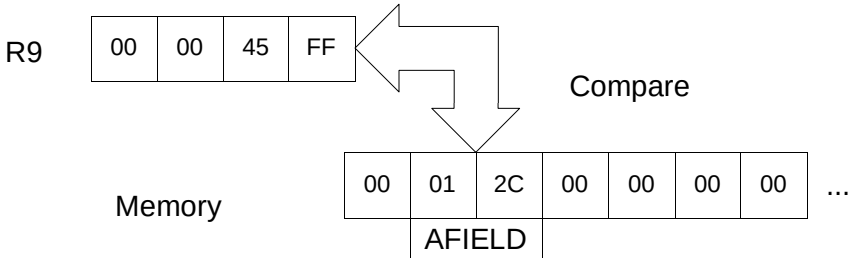
```
R4 = X'FFFFFFD5' -43 IN 2'S COMPLEMENT
R5 = X'00000028' +40 IN 2'S COMPLEMENT
R6 = X'00000004' +4 IN 2'S COMPLEMENT

      DOG      DC   F'35'
      CAT      DC   F'4'

               C    R4,=F'20'      CONDITION CODE = LOW
               C    R5,=F'-20'     CONDITION CODE = HIGH
               C    R6,=F'20'      CONDITION CODE = LOW
               C    R6,=F'4'       CONDITION CODE = EQUAL
               C    R5,=F'40'      CONDITION CODE = EQUAL
               C    R5,DOG         CONDITION CODE = HIGH
               C    R6,DOG(R6)     CONDITION CODE = EQUAL
```

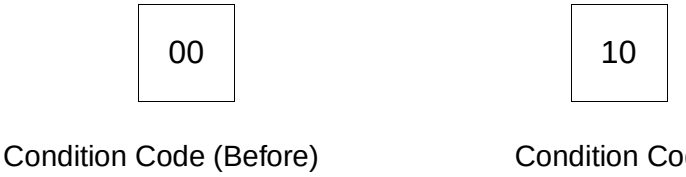
Op Code	R_1X_2	B_2D_2	D_2D_2

The Compare Halfword instruction (**CH**) is used to compare a binary fullword in a register, Operand 1, with a binary halfword in memory, Operand 2. The operands are compared as 2's complement signed binary integers. For purposes of comparison, the halfword is "sign-extended" to a fullword before the comparison occurs. This extension occurs internally and is temporary. The instruction sets the condition code to indicate how Operand 1 compares to Operand 2:

Condition Code	Meaning	Test With
0	Operand 1 = Operand 2	BE, BZ
1	Operand 1 < Operand 2	BL, BM
2	Operand 1 > Operand 2	BH, BP

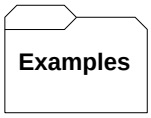
```
CH R9,AFIELD
```





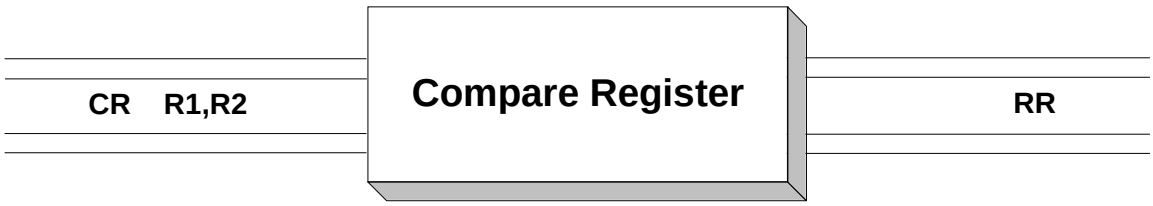
The contents of the halfword "AFIELD", x'012C', is sign-extended, and is compared to the contents of register 9 which contains x'000045FF'. Since the contents of the register (Operand 1) is greater than the value than the extended halfword (Operand 2), the condition code is set to "High". The condition code in the diagram above is specified using 2 binary digits. After comparison, the condition code is set to a binary 10 which is 2 in decimal - a "High" condition.

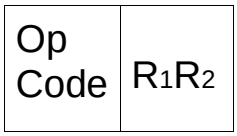
Since **CH** is an RX instruction, an index register may be coded as part of operand 2.



Some Unrelated Compare Halfwords

```
R4 = X'FFFFFFD5' -43 IN 2'S COMPLEMENT
R5 = X'00000028' +40 IN 2'S COMPLEMENT
R6 = X'00000004' +4 IN 2'S COMPLEMENT

            DOG      DC   H'40'
            CAT      DC   H'-30'
            PIG      DC   H'14'
            GOAT     DC   H'3'

                     CH   R4,=H'20'    CONDITION CODE = LOW
                     CH   R4,=H'-50'   CONDITION CODE = HIGH
                     CH   R5,=H'20'    CONDITION CODE = HIGH
                     CH   R6,=H'4'     CONDITION CODE = EQUAL
                     CH   R5,=H'40'    CONDITION CODE = EQUAL
                     CH   R5,DOG       CONDITION CODE = EQUAL
                     CH   R6,DOG(R6)   CONDITION CODE = LOW
                                       DOG(R6) IS EQUIVALENT TO PIG
```



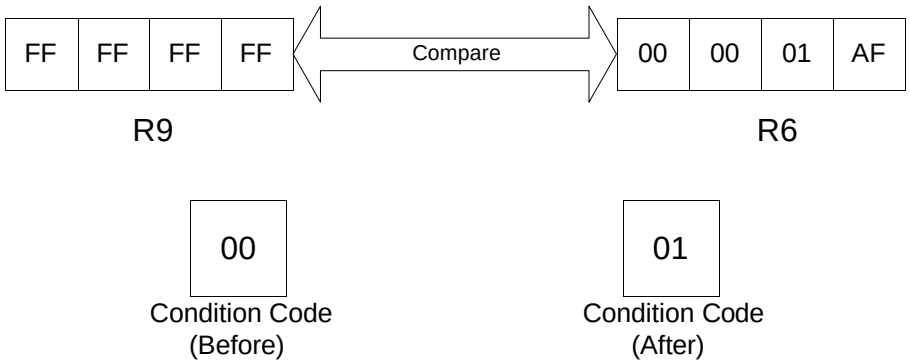


The Compare Register (**CR**) instruction is used to compare a binary fullword in a register, designated by Operand 1, with another fullword in a register, designated by Operand 2. The operands are compared as 32-bit signed binary integers. The instruction sets the condition code to indicate how Operand 1 compares to Operand 2:

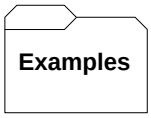
Condition Code	Meaning	Test With
0	Operand 1 = Operand 2	BE, BZ
1	Operand 1 < Operand 2	BL, BM
2	Operand 1 > Operand 2	BH, BP

The following example sets the condition code by comparing registers 9 and 6.

```
CR R9,R6
```

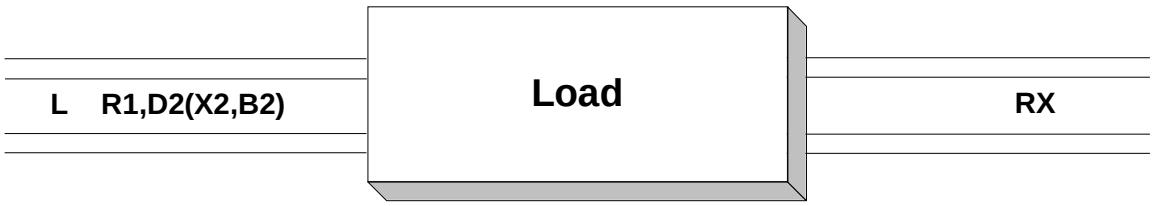


The contents of the fullword in register 9, x'FFFFFFFF' = -1, is compared to the contents of register 6 which contains x'000001AF' = 431. Since the contents of the Operand 1 register is less than the contents of the Operand 2 register, the condition code is set to "Low". The condition code in the diagram above is specified using 2 binary digits. After comparison, the condition code is set to a binary 01 which is 1 in decimal - a "Low" condition.



Some Unrelated Compare Register Instructions

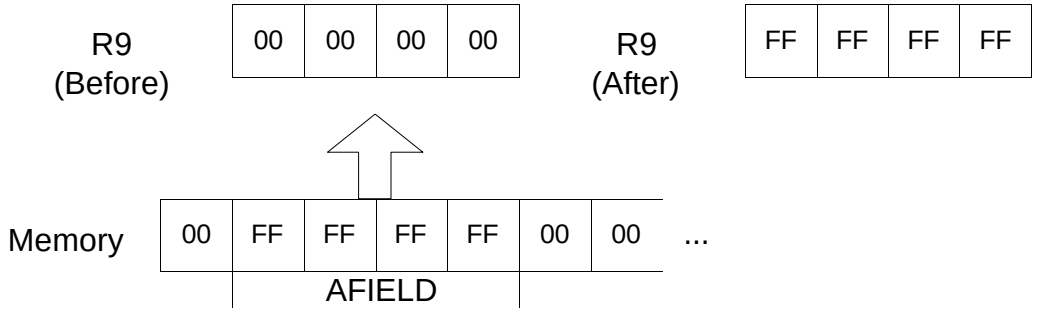
```
R4 = X'FFFFFFD5' -43 IN 2'S COMPLEMENT
R5 = X'00000028' +40 IN 2'S COMPLEMENT
R6 = X'00000004' +4 IN 2'S COMPLEMENT

           CR   R4,R5   CONDITION CODE = LOW
           CR   R5,R4   CONDITION CODE = HIGH
           CR   R4,R4   CONDITION CODE = EQUAL
           CR   R6,R5   CONDITION CODE = LOW
           CR   R5,R5   CONDITION CODE = EQUAL
```



Op Code	R_1X_2	B_2D_2	D_2D_2

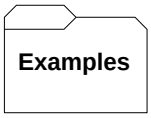
The Load **(L)** instruction is used to copy the fullword stored in the memory location designated by operand 2 into the register specified by operand 1. Consider the following example,

```
L R9,AFIELD
```



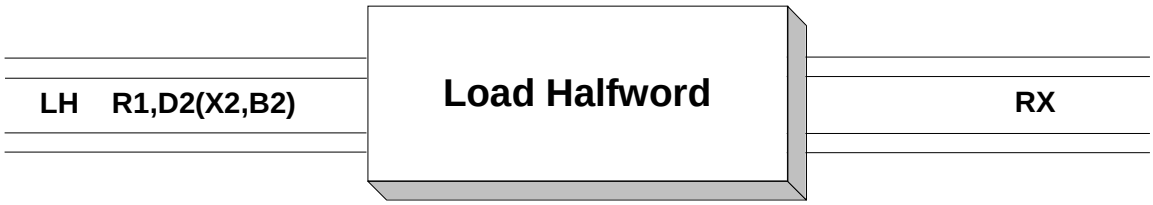
The contents of the fullword "AFIELD" are copied to register 9, destroying the previous values in R9. The fullword is unchanged by this operation.

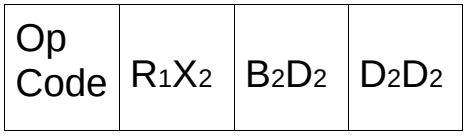
Since **L** is an RX instruction, an index register may be coded as part of operand 2.



Some Unrelated Loads

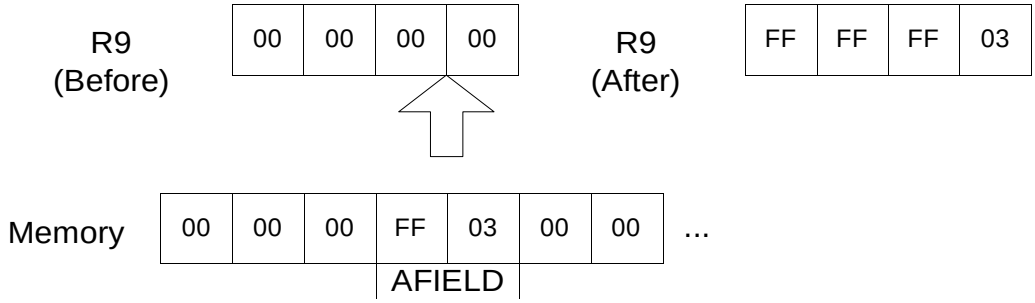
```
R4 = X'12121212'
R5 = X'00000008'
R6 = X'00000004'

AFIELD   DC   F'4'          AFIELD = X'00000004'
BFIELD   DC   F'-1'         BFIELD = X'FFFFFFFF'
CFIELD   DC   F'0'          CFIELD = X'00000000'

         L    R4,AFIELD     R4 = X'00000004'
         L    R4,AFIELD(R6) R4 = X'FFFFFFFF'
         L    R4,AFIELD(R5) R4 = X'00000000'
         L    R6,AFIELD(R6) R6 = X'FFFFFFFF'

         CONSIDER THE NEXT TWO CONSECUTIVELY EXECUTED LOADS
         L    R5,AFIELD     R5 = X'00000004'
         L    R6,AFIELD(R5) R6 = X'FFFFFFFF'
```





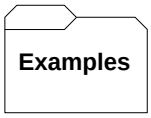
The Load Halfword (**LH**) instruction has two operands: the first operand is a general-purpose register, and the second operand is a halfword in memory. The effect of the instruction is to place a 32-bit signed integer, algebraically equivalent to the 16-bit halfword, into Operand 1. The 32-bit integer can be obtained by extending the sign-bit of the halfword so that it occupies 32 bits. (Extending the sign bit of 2's complement data does not change its arithmetic value.) Consider the following example.

```
LH R9,AFIELD
```



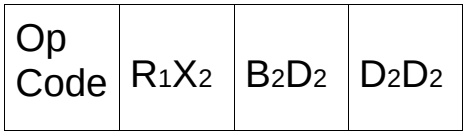
The halfword "AFIELD" contains a binary 1 in the sign-bit. A sign-extended version of the halfword is copied into register 9, destroying the previous values in the register. The halfword in AFIELD is unchanged by this operation.

Since **LH** is an RX instruction, an index register may be coded as part of operand 2.



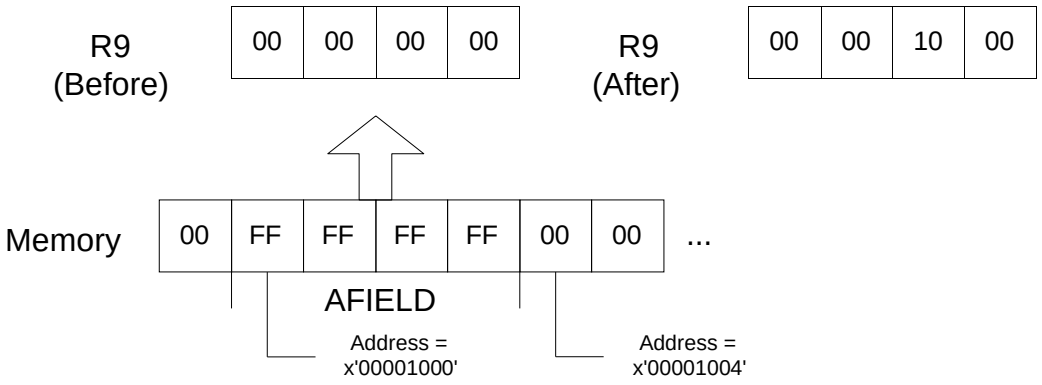
Some Unrelated Load Halfword Instructions

```
R4 = X'00000000'
R5 = X'FFFFFFFF'
R6 = X'00000004'

    AFIELD   DC   H'4'          AFIELD = X'0004'
    BFIELD   DC   H'-1'         BFIELD = X'FFFF'
    CFIELD   DC   H'0'          CFIELD = X'0000'

             LH   R4,AFIELD     R4 = X'00000004'
             LH   R4,BFIELD     R4 = X'FFFFFFFF'
             LH   R4,CFIELD     R4 = X'00000000'
             LH   R4,DFIELD     R4 = X'FFFF

             CONSIDER THE NEXT TWO CONSECUTIVELY EXECUTED LOADS
             LH   R5,AFIELD     R5 = X'00000004'
             LH   R6,AFIELD(R5) R6 = X'FFFFFFFF'
```



The Load Address (**LA**) instruction is used to initialize the register specified by operand 1 with the address of operand2. Operand 2 may be expressed using explicit notation or symbolic notation, or a combination of both. Remember that each byte in memory is numbered and that the number assigned to a byte is its address. The address of a field is the address of the first byte of the field. Consider the following example,

```
LA R9,AFIELD
```



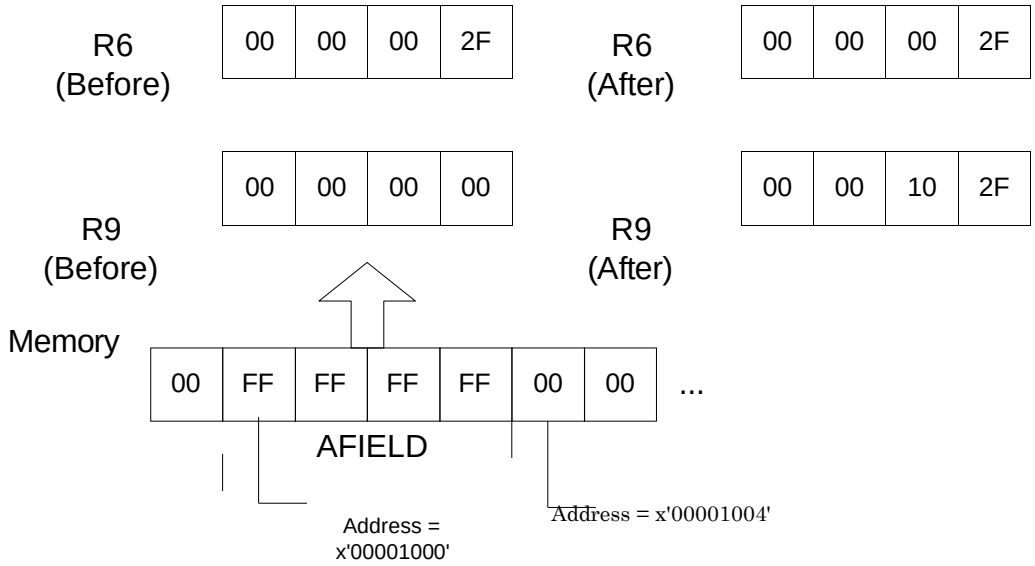
The address of the fullword "AFIELD", x'00001000', is copied to register 9, destroying the previous value in R9. The fullword is unchanged by this operation.

Since **LA** is an RX instruction, an index register may be coded as part of operand 2 as in the example below. We assume that register 6 is used as an index register and initially contains x'0000002F'. When the assembler processes the expression AFIELD(R6), it uses the symbol AFIELD to determine a base register and a displacement, leaving R6 as the index register. The address which is loaded into register 9 is the "effective address" computed by adding the base register contents, plus the index register contents, plus the displacement:

Effective address = C(Base register) + C(Index register) + displacement

If we assume that the contents of the base register plus the displacement is x'00001000'. Then the effective address is x'00001000' + x'0000002F' = x'0000102F'.

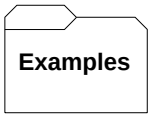
```
LA R9,AFIELD(R6)
```



The example above uses a mixture of symbolic and explicit addressing. The instruction could also be coded using only explicit addresses:

```
LA R9,30(R7,R8)
```

In the example above assume that R7 contains x'00001000' and that R8 contains x'00000020'. R7 is treated as an index register, R8 is the base register, and 30 is a displacement. The effective address is C(R7) + C(R8) + 30 = x'00001000" + x'00000020' + x'0000001E' = x'0000103E'. (Remember that a decimal 30 is 1E in hexadecimal.) After the instruction has executed, R9 contains x'0000103E'.



Some Unrelated Load Address Instructions

```
R4 = X'12121212'
R5 = X'00000008'
R6 = X'00000004'
```

Assume that AFIELD has address x'00003000'.

```
AFIELD   DC  F'4'            AFIELD = X'00000004'

         LA  R4,AFIELD       R4 = X'00003000'
         LA  R4,AFIELD(R6)   R4 = X'00003004'
         LA  R4,AFIELD(R5)   R4 = X'00003008'
         LA  R4,20(R5,R6)    R4 = X'00000020' 4 + 8 + 20 = 32 = X'20'
```

Using R0 as an index indicates that no index register is desired:

```
         LA  R4,3(R0,R6)        R4 = X'00000007' 4 + 3 = 7
```

Consider the next two consecutively executed instructions.

```
         LA  R4,AFIELD          R4 = X'00003000'
         LA  R4,L'AFIELD(R0,R4) R4 = x'00003004'
```

In the example above, the length attribute (L') is used as a displacement.

Tips

1. An old assembler joke:

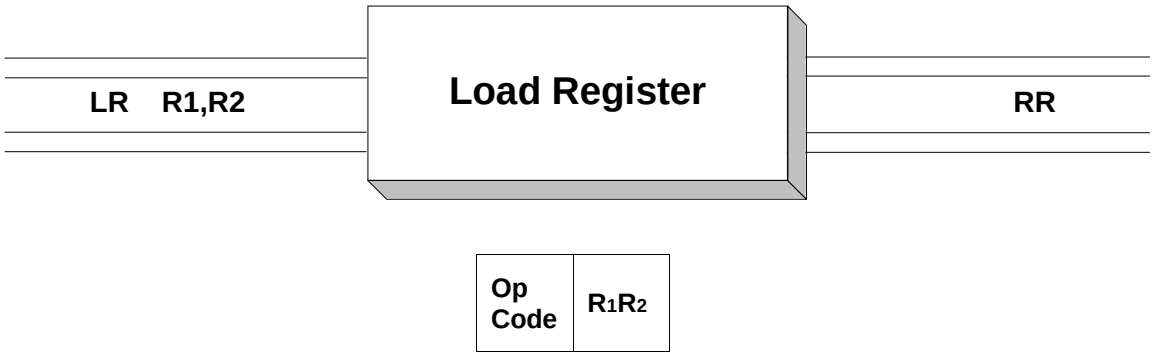
Novice: What's the difference between a Load instruction and a Load Address instruction?"

Old Hand: About a week … of debugging.

Seriously, you should pay attention when coding **L** or **LA**. Both instructions compute the address of operand 2. In the case of **L**, the machine retrieves the contents of the fullword in memory at the specified address and places the four bytes in a register. In the case of **LA**, the address is simply stored in a register.

2. The **LA** instruction is often used to change the location referenced by a DSECT:

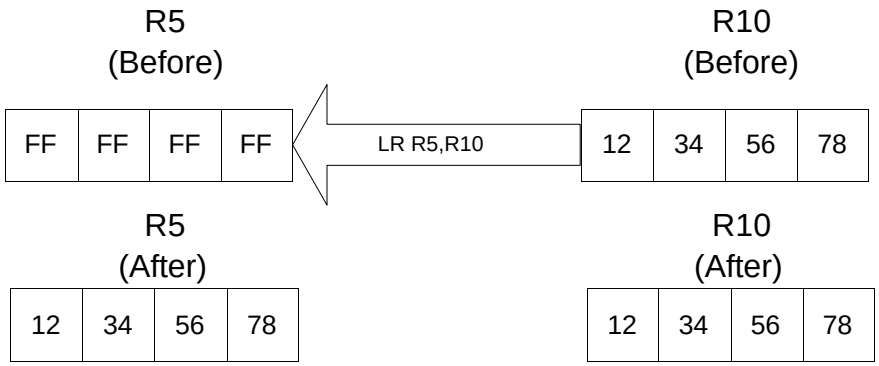
```
TEST     DSECT
TESTREC  DS 0CL80
X        DS ...

         USING TEST,R5
         ...
         LA   R5,TABLE               POINT AT TABLE AREA ...
         LA   R5,L'TESTREC(R0,R5)    MOVE THE DSECT
```



The Load Register (**LR**) instruction copies the contents of the register specified by Operand 2, into the register specified by Operand 1. The contents of Operand 2 are unchanged as well as the condition code. Consider the instruction below.

```
LR R5,R10
```

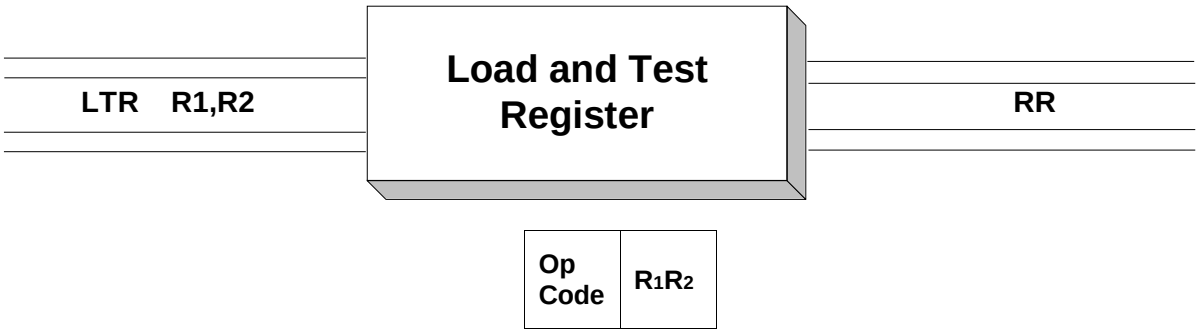
The contents of register 10 are copied to register 5, destroying the previous value in register 5. Register 10 is unaffected by the operation. The diagram below illustrates this operation.



Examples

Some Unrelated LR's

```
R4 = X'FFFFFFFF'
R5 = X'00000028'
R6 = X'00000004'

            LR   R4,R5          R4 = X'00000028' R5 = X'00000028'
            LR   R5,R4          R5 = X'FFFFFFFF' R4 = X'FFFFFFFF'
            LR   R5,R6          R5 = X'00000004' R6 = X'00000004'
            LR   R6,R5          R6 = X'00000028' R5 = X'00000028'
```



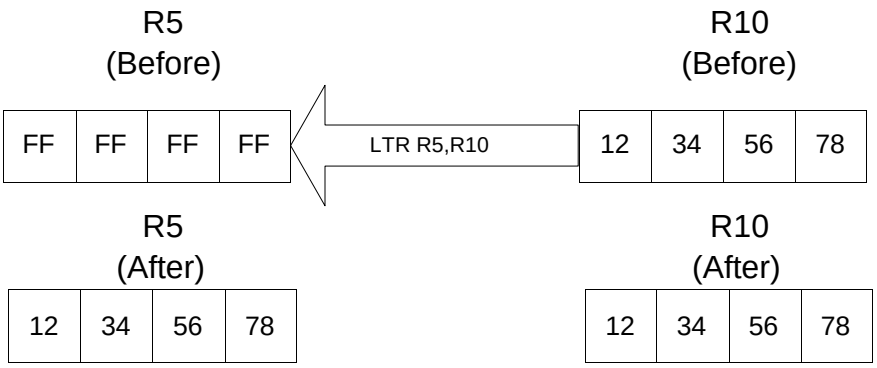
The Load and Test Register (**LTR**) instruction copies the contents of the register specified by Operand 2, into the register specified by Operand 1. The contents of Operand 2 are unchanged by this operation. In this respect **LTR** is equivalent to the **LR** instruction. The difference between these instructions is that **LTR** sets the condition code based on the final contents of the Operand 1 register.

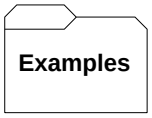
Condition Code	Meaning	Test With
0 (Zero)	Operand 1 = 0	BE, BZ
1 (Negative)	Operand 1 < 0	BL, BM
2 (Positive)	Operand 1 > 0	BH, BP

Consider the instruction below.

```
LTR   R5,R10
```

The contents of register 10 are copied to register 5, destroying the previous value in register 5. Register 10 is unaffected by the operation. Since the contents of R5 is positive after completion of the operation, the condition code is set to 2. The diagram below illustrates this operation.





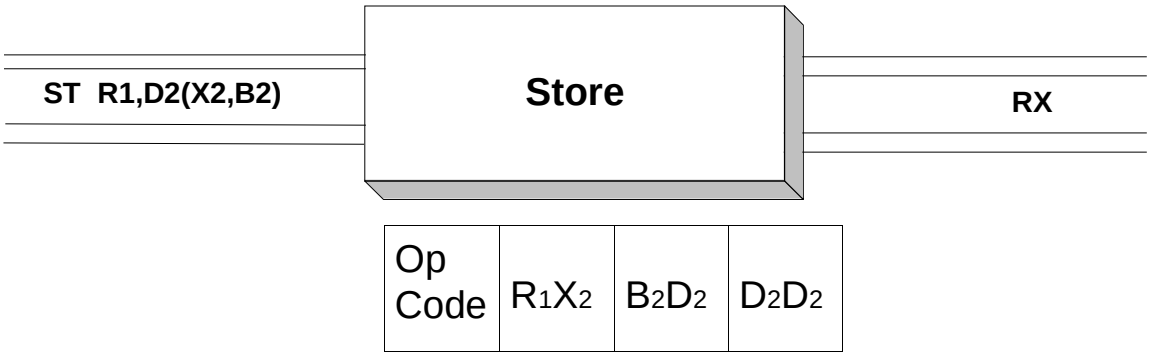
Some Unrelated LTR Instructions

```
R4 = X'FFFFFFFF'
R5 = X'00000028'
R6 = X'00000004'
R7 = X'00000000'

    LTR  R4,R5      R4 = X'00000028' R5 = X'00000028' Cond. Code = Positive
    LTR  R5,R4      R5 = X'FFFFFFFF' R4 = X'FFFFFFFF' Cond. Code = Negative
    LTR  R5,R6      R5 = X'00000004' R6 =  X'00000004' Cond. Code = Positive
    LTR  R6,R5      R6 = X'00000028' R5 =  X'00000028' Cond. Code = Positive
    LTR  R6,R7      R6 = X'00000000' R7 =  X'00000000' Cond. Code = Zero
    LTR  R4,R4      R4 = X'FFFFFFFF' Cond. Code = Negative
```

Tips

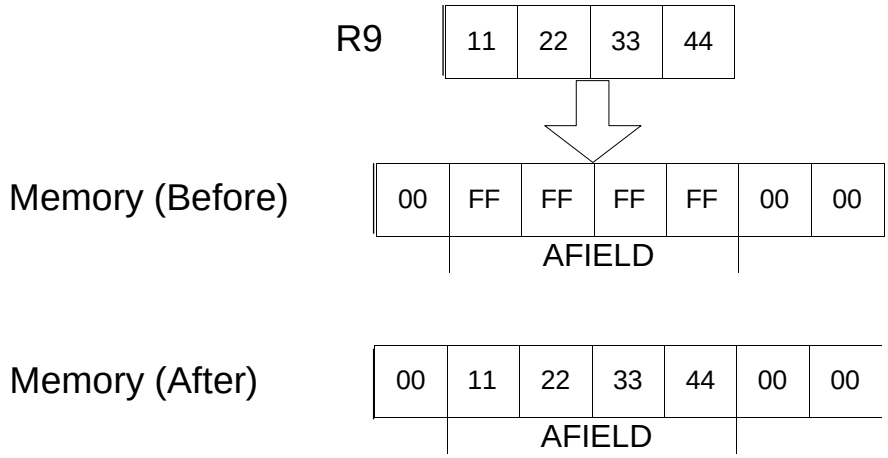
1) **LTR** is commonly used to test the contents of a single register to determine if the binary number in the register is positive, negative or zero. For example, the following code illustrates how to test the contents of register 5.

```
          LTR  R5,R5            SET THE CONDITION CODE
          BM   NEGATIVE         IS R5 < 0 ?
          BP   POSITIVE         IS R5 > 0?
ZERO      EQU  *
          ...
NEGATIVE  EQU *
          ...
POSITIVE  EQU *
```



The Store instruction (**ST**) is used to copy the fullword stored in the register specified by operand 1 into the fullword memory location specified by operand 2. Consider the following example,

```
ST R9,AFIELD
```



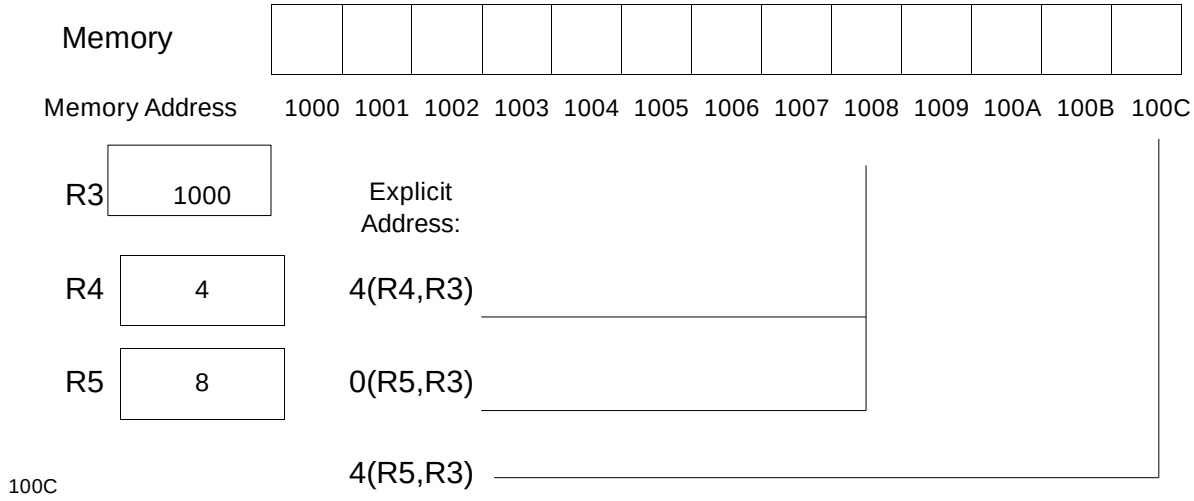
In this case, the contents of register 9 are copied to the fullword in memory denoted by AFIELD. This operation destroys the previous contents of AFIELD but leaves R9 unchanged.

Since **ST** is an RX instruction, an index register may be coded as part of operand 2. Notice that in the previous example, no index register was specified. When the index register is omitted, the assembler chooses R0, which does not contribute to the address. The following example illustrates this idea.

```
ST R9,AFIELD(R5)
```

The assembler converts AFIELD to a base register and displacement, while R5 is the index register. For instance, the expression AFIELD(R5) might (we cannot determine the base register with limited information) be equivalent to the explicit address 0(R5,R3) - displacement = 0, index register = R5, base register = R3. The effective address is computed by adding the base register contents to the index register contents plus the displacement. If the index register contains an "8", then AFIELD(R5) refers to the fullword that begins at an 8-byte displacement from the beginning byte of AFIELD. The following examples

illustrate several explicit addresses that include an index register.



In the first explicit address, 4(R4, R3), the effective address is computed by adding the contents of base register 4, the contents of index register 3, and the displacement (1000 + 4 + 4 = 1008). The second address 0(R5, R3) is computed as 1000 + 8 + 0 = 1008, and the third address, 4(R5, R3) is computed to be 1000 + 8 + 4 = 100C (hexadecimal).

If an index register is not explicitly coded, as in the instruction "ST R9,AFIELD", the assembler chooses R0 as the index register, which does not contribute to the effective address.

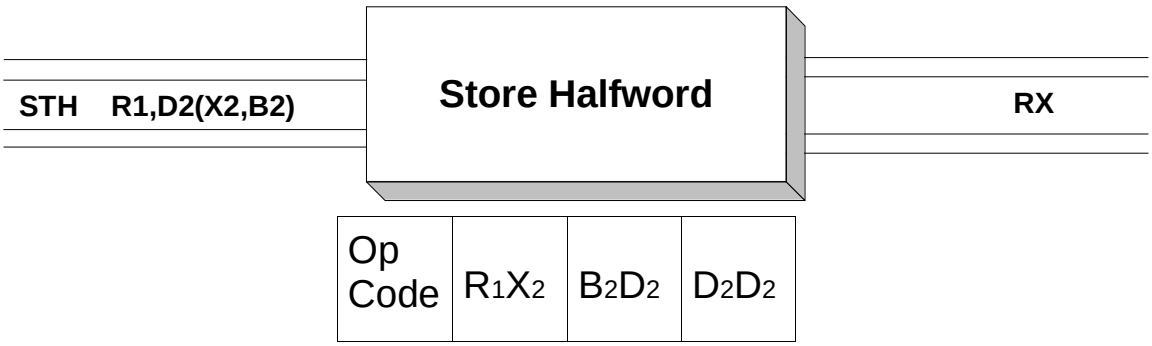
Examples

Some Unrelated ST's

```
R7 = X'00001000'R8 = X'00000004' R9 = X'00000008'

        AFIELD   DC F'20'          AFIELD =  X'00000016'
        BFIELD   DC F'-1'          BFIELD =  X'FFFFFFFF'
        CFIELD   DC F'0'           CFIELD =  X'00000000'

                 ST R7,AFIELD      AFIELD =  X'00001000'
                 ST R8,AFIELD      AFIELD =  X'00000004'
                 ST R8,BFIELD      BFIELD =  X'00000004'
                 ST R7,AFIELD(R8)  CHANGES BFIELD TO X'00001000'
                 ST R7,AFIELD(R9)  CHANGES CFIELD TO X'00001000'
```

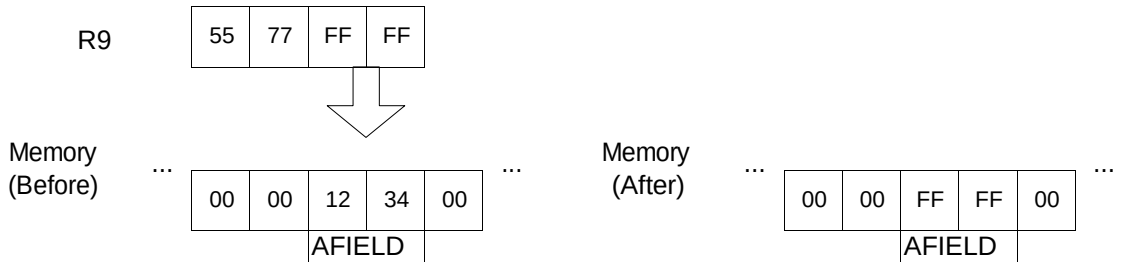
Tips

1) Operand 2 should denote a fullword in memory. It is possible to store the contents of a register into 4 bytes of memory that are not aligned on a fullword, but the assembler will warn you that operand 2 is not properly aligned. If the field involved cannot be aligned conveniently, consider using **STCM** to copy the contents of a register into memory.



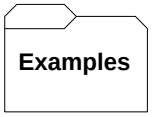
The Store Halfword (**STH**) instruction has two operands: the first operand is a general-purpose register, and the second operand is a halfword storage area in memory. The effect of the instruction is to copy the contents of bits 16-31 of the Operand 1 register (the rightmost two bytes) into the halfword specified by Operand 2. The condition code is unaffected by this instruction. Consider the following example.

```
STH  R9,AFIELD
```



The rightmost two bytes of register 9 are copied to the halfword AFIELD, destroying the previous contents. The register value is unchanged by this operation.

Since **STH** is an RX instruction, an index register may be coded as part of operand 2.



Some Unrelated Store Halfword Instructions

```
R4 = X'00000000'
R5 = X'12345678'
R6 = X'00000004'


            AFIELD   DC   H'4'           AFIELD = X'0004'
            BFIELD   DC   H'25'          BFIELD = X'0019'
            CFIELD   DC   H'0'           CFIELD = X'0000'

                     STH  R4,AFIELD      AFIELD = X'0000'
                     STH  R4,BFIELD      BFIELD = X'0000'
                     STH  R4,CFIELD      CFIELD = X'0000'
                     STH  R5,AFIELD      AFIELD = X'5678'
                     STH  R6,BFIELD      BFIELD = X'0004'

CONSIDER THE NEXT TWO CONSECUTIVELY EXECUTED INSTRUCTIONS

                     LH   R8,AFIELD      R8 = X'00000004'
                     STH  R5,AFIELD(R8)  BFIELD = X'5678'
```

Tips

1) Consider using **STCM** if you need to store a halfword in memory. It has the advantage of being able to copy the value to memory without taking alignment into consideration.

Programming Exercises

1) Create a file with the following ten records in character/zoned decimal format in a member in your PDS. For safety, keep the file in a PDS that is separate from you code, or if you have the skills to create the file as a standalone file outside of any PDS, you can do that. We just need a readable file with this data.

```
0000000006000000000400000000003
0000000012000000003200000000064
0000000012000000007200000000000
0000000005000000002000000000003
0000000228000000008700000000032
0000000012000000001200000000050
0000000256000000025600000000388
0000002592000000043200000002592
0000003000000000040000000000028
0000000010000000003000000002000
```

Each record contains three integer fields:

```
A – Columns 1-10
B – Columns 11-20
C – Columns 21-30
```

Read the file you just created and use the records to build an QSAM output file. Write an 80-byte output record for each input record. Each output record has the following format:

```
Columns 1-4   – Field A stored as a binary fullword
Columns 5-6   – Field B stored as a binary fullword
Columns 7-8   – Field C stored as a binary halfword
Columns 9-19  – Unused
Columns 20-29 – Field A in an edited character format
Columns 30-39 – Field B in an edited character format
Colunns 40-49 – Field C in an edited character format
Columns 50-80 – Unused
```

2) Read the output file that was created in 1) above. Print a report that looks like the one below. Formatting can vary – make it look presentable. Perform all the arithmetic in the registers.

```
      A            B           C        A + B – C     (A * B)/ C
-------------------------------------------------------------------
      2            4           8           2-            1
     12            8           7          13             5
      1            2           3           0             0
     20           30          80          30-            7
      2            4           6           0          ****
```

Hints: Start by creating a report that prints the first three columns only. Verify you are correctly reading the input data, then add the next two columns one at a time. You will need to align the output record properly to avoid assembly warnings.

PROBLEMS

9-1. Here is a 32-bit 2's complement integer – X'00000012'. What number is that in decimal?

9-2. Here is a 32-bit 2's complement integer – X'FFFFFF12'. What number is that in decimal?

9-3. Here is a 16-bit 2's complement integer – X'FF12'. What number is that in decimal?

9-4. Here is a 16-bit 2's complement integer – X'00AB'. What number is that in decimal?

9-5. Here is a 16-bit 2's complement integer – X'FFAB. What does the 32-bit 2's complement version of that number look like?

9-6. Assume the following declaration,

```
X          DS     ZL5
```

Write the code that will convert X to a binary number in register 5. Define any fields you need.

9-7. Assume the following declaration,

```
Y          DS     ZL10
```

Write the code that would convert the contents of register 7 into Y. Define any fields you need.

9-8. Assume A and B are fullwords in memory. Write the code that will compute the sum of A and B in register 8.

9-9. Assume A and B are fullwords in memory. Write the code that will compute the quotient of A and B. Leave the quotient in register 7. Be sure not to divide by zero.

9-10. Which data format can store a larger range of integers, binary fullwords or packed decimal integers?