

Assembler University 207: Powerful New z/Architecture Instructions That Don't Require AMODE(64), Part 2

SHARE 116 in Anaheim, Session 8983

Avri J. Adleman, IBM

adleman@us.ibm.com

(Presented by John Ehrman, IBM)

March 3, 2011

- Shifting Instructions
 - 64-bit shifting
 - Rotate
- Packed Decimal Instructions
 - Test Packed
 - CVB and CVD enhanced instructions
 - Pack and Unpack ASCII
- Translate (and Test) Instructions
 - TRTR
 - TRE and TR_{xx}
 - TRTE and TRTRE

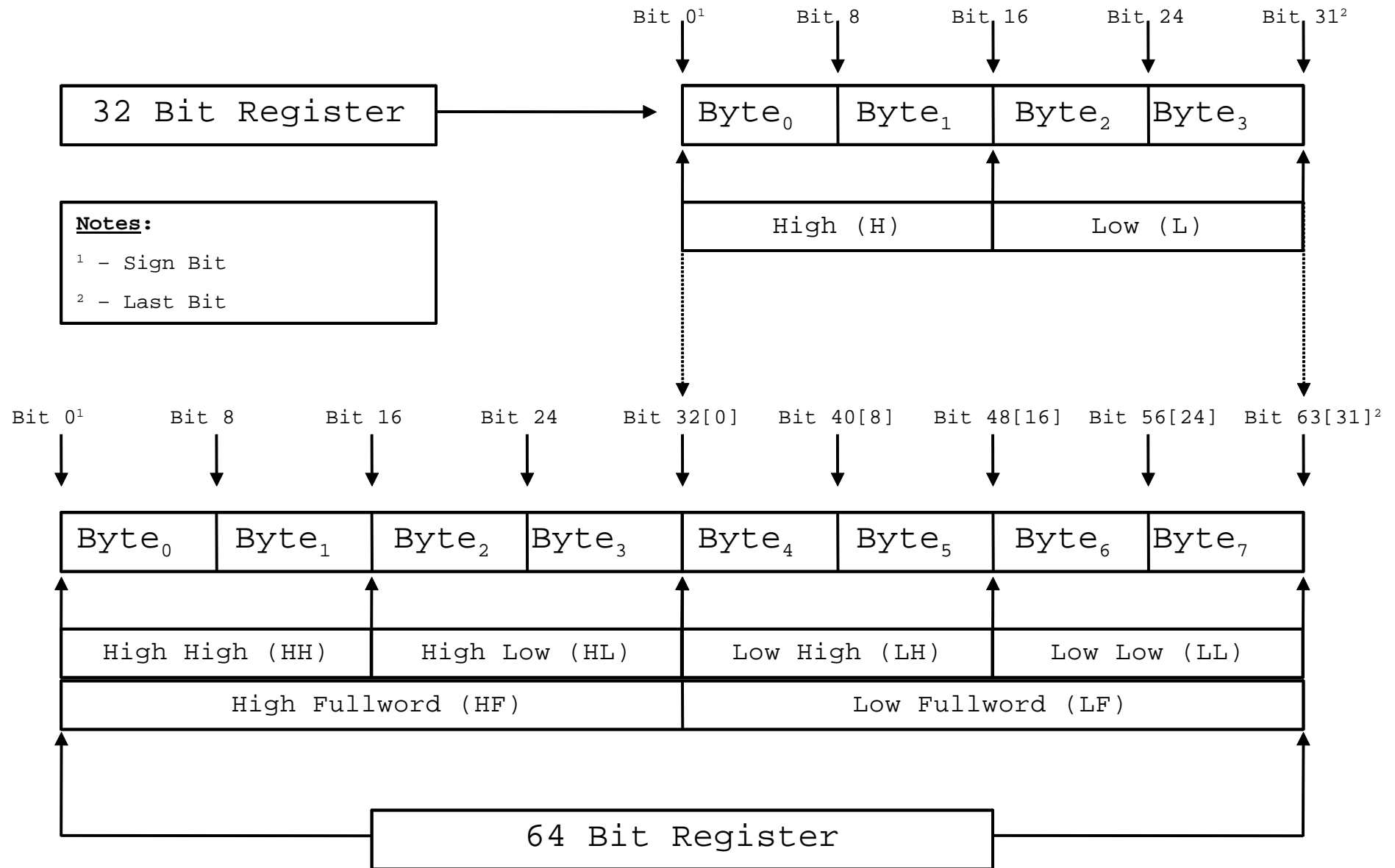
Terminology: all machine generations



Byte	8 bits
Halfword	2 Bytes (16 Bits)
Fullword (Word)	4 Bytes (32 Bits)
Doubleword	8 Bytes (64 Bits)
Quadword	16 Bytes (128 Bits)

- **Notation:** *64-bit based* [*32-bit based*]
 - 64-bit based (Doubleword)
 - 32-bit based (Fullword)
- **Positions:**
 - “*High Order*” refers to the low numbered bits
 - “*Low Order*” refers to the high numbered bits

Register Layout



Instruction mnemonic usage



Mnemonic	Name	Instruction Examples	Additional Remarks
LL????	Load Logical	LLGT, LLGC, LLGH, ...	Loads specific bytes of a register, fills remainder with zeroes.
??G??	Grande Register	LGR, AG, LTGR, ...	Applies to full 64-Bit register as target or target and source; may widen value with or without sign propagation.
??F??	Fullword ("traditional register")	LGF, LGFR, ALGF, ...	Applies to 32-bit word as source; value is widened when target is a 64-bit register.
??T??	Thirty-One Bit	LLGTR, LLGT	Applies to source as the lower 31 bits: bit 33[1] to bit 63 [31]
??H??	Halfword (2 bytes)	LGH, AGH, ...	Applies to a halfword (a pair of specified bytes) of a 64 bit register.
??H??	High word of a 64-bit register	LMH, STMH	Applies to the high word, bits 0 to 31, of a 64 bit register
????LL, ????LH, ????HL, ????HH	Low-Low Low-High High-Low High-High	TMLL, LLIHH, ...	Specified halfwords of a 64-bit register
II????	Insert-Immediate	IILL, IILH, ...	Load specific bytes of a register, leaving remainder alone.

64-bit “Grande” shift instructions



- Allowable shifts
 - Right vs. Left
 - Arithmetic (CC set) vs. Logical (CC unchanged)
 - Shift amount determined by the six bit effective address
 - 0 to 63 bits
 - **No** double (even/odd pair, 128-bit) shifting of 64-bit register pairs
- Entire 64-bit register partakes in the shifting
 - Allows different target and source registers
 - **SxyG R₁, R₃, D₂(B₂)**
 - R₁ is target, R₃ is source
 - Enables retention of original operand value
- Instructions
 - SLLG, SRLG, SLAG, SRAG

Shifting examples

* Example #1:

SLLG R2,R1,4

* Before: R1 = X'000000FFFFFFFFF'

* Before: R2 = ?

* After: R1 = X'000000FFFFFFFFF'

* After: R2 = X'00000FFFFFFFFF0'

* Example #2:

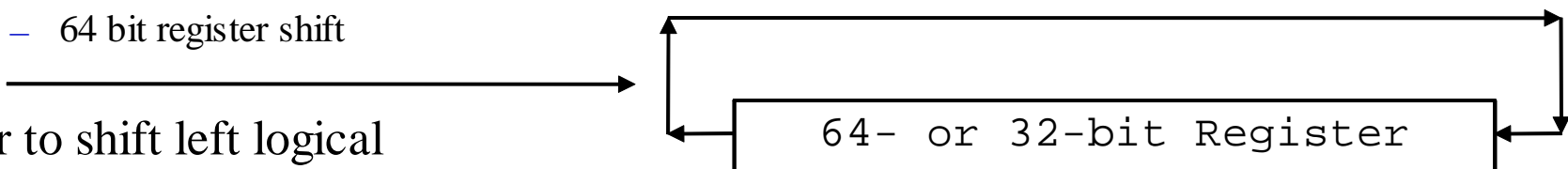
SRAG R1,R1,5

JM WILLBRNH

* Before: R1 = X'8000000000000000'

* After: R1 = X'FC00000000000000'

Rotate bits in a register

- Rotate Left Single Logical
 - Also known as circular shift
 - Instructions (R_1 is the target and R_3 is the source)
 - **RLL** $R_1, R_3, D_2(B_2)$
 - 32 bit register shift
 - **RLLG** $R_1, R_3, D_2(B_2)$
 - 64 bit register shift
- Process 
 - Similar to shift left logical
 - Shifting to the left
 - Shift amount determined by base and displacement: $D_2(B_2)$
 - No condition code change, overflow not recognized
 - Carry-out bits are shifted back into the low order bits
 - No loss of bits
 - Target register of shift can differ from source register
- Special Notes:
 - No Rotate Right Single Logical Instruction
 - Use shift factor of 32 or 64 minus shift amount on **RLL** or **RLLG** instruction

Rotate examples

* Example #1:

RLLG R2,R1,4

* Before: R1 = X'**F**0000000000000000'

* Before: R2 = ?

* After: R1 = X'**F**0000000000000000'

* After: R2 = X'0000000000000000**F**'

* Example #2:

RLLG R1,R1,32

* Before: R1 = X'**FFFFFFFF**00000000'

* After: R1 = X'00000000**FFFFFFFF**'

* Example #3:

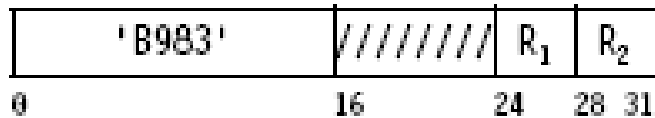
RLL R1,R1,1

* Before: R1 = X'**AABBCCDD8**00000000'

* After: R1 = X'**AABBCCDD0**0000000**1**'

FLOGR Instruction

- **FLOGR** R_1, R_2 [RRE]

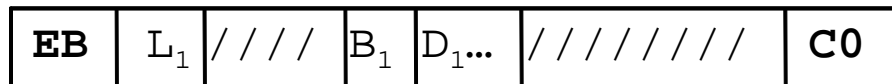


- Find Leftmost One Bit “*Grande*” Register (FLOGR)
 - Register R_1 must be an even-odd pair
 - Register R_2 any single 64-bit register
- Scan 64-bit register (R_2) left to right to find first one bit
 - If found:
 - Set register (R_1) with 0 based bit index (0 to 63) of first one bit in (R_2)
 - R_2 is copied into R_1+1 with found bit set to 0
 - Condition Code set to 0
 - If not found:
 - Set register (R_1) to 64
 - Set register R_1+1 to 0
 - Condition Code set to 2

Test Packed (Decimal)

- Test Decimal Instruction

- **TP** $D_1, (L_1, B_1)$



- RSL format

- D_1 and B_1 are the base and displacement of the testing value
 - L_1 represents the length of the field ranging from 1 to 16

- Extended Translation Facility 2

- Tests argument for a valid packed number format

- No need to use complex coding such as:

- Scanning digits and sign via looping
 - Setting ESTAE and/or ESPIE exits

- Sets the condition code

- CC = 0: All digits and sign codes are valid
 - CC = 1: Sign is invalid
 - CC = 2: At least one digit code is invalid
 - CC = 3: Sign invalid and at least one digit is invalid

* <u>Test Decimal Example</u>		
TP	PACKED	
JNZ	INVALID	
. . .		
INVALID DS	OH	
. . .		
PACKED DC	P'nnnn'	

Binary and Packed extended conversion

- Convert to Binary Extensions
 - **CVBY** R_1 ,DBLWRD
 - Same as CVB, but with extended displacement
 - R_1 is a 32 bit register
 - DBLWRD is eight bytes with RXY storage and packed data
 - **CVBG** R_1 ,QUADWRD
 - Similar to CVB
 - R_1 is a 64 bit register
 - QUADWRD is sixteen bytes with RXY storage and packed data
 - Supports extended displacement
- Convert to Decimal Extensions
 - **CVDY** R_1 ,DBLWRD
 - Same as CVD, but with extended displacement
 - R_1 is a 32 bit register
 - DBLWRD is eight bytes with RXY storage and packed data
 - **CVDG** R_1 ,QUADWRD
 - Similar to CVB
 - R_1 is a 64 bit register
 - QUADWRD is sixteen bytes with RXY storage and packed data
 - Supports extended displacement

- **PKA** $D_1(B_1), D_2(L_2, B_2)$ [SS format]
 - Similar to the PACK instruction
 - Target (Arg_1) has zone data removed from source (Arg_2)
 - Numeric field of source transferred byte to digit
 - Implied positive sign is placed in rightmost hex digit of target
 - ASCII, with a binary value of 1100
 - Note exceptions:
 - First argument $D_1(B_1)$
 - *Length always 16*
 - Zero filled if necessary
 - Second argument $D_2(L_2, B_2)$ has variable length
 - Ranges 1 to 32 in length
 - Greater than 32 is an exception
 - Equal to 32 is allowed, but leftmost byte is ignored

Unpack ASCII



- **UNPKA $D_1(L_1, B_1), D_2(B_2)$ [SS format]**
 - Similar to the UNPK instruction
 - Expands packed BCD digits in source (Arg_2) into ASCII numeric characters (**$x'30'$** to **$x'39'$**) in target (Arg_1)
 - Note exceptions:
 - No transfer of sign
 - Instead the condition code is set based on sign:
 - 0 = plus, 1 = minus, 3 = invalid packed decimal sign
 - First Argument $D_1(L_1, B_1)$
 - Ranges in length from 1 to 32 bytes
 - Greater than 32 is an exception
 - If length is too small, leftmost digits truncated
 - Second Argument $D_2(B_2)$ *length is always 16*

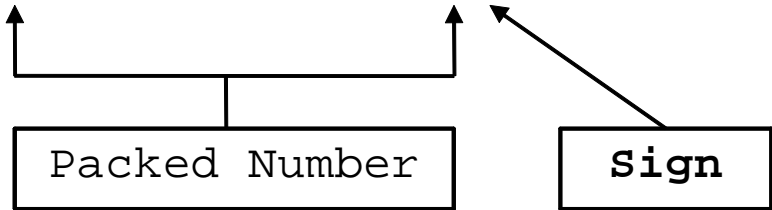
More on PKA and UNPKA



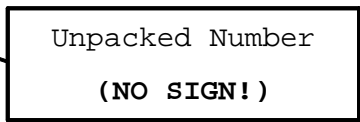
- Usage with EBCDIC
 - Instructions do not raise a data exception
 - PKA can be used on EBCDIC or non-ASCII data
 - Source need not contain valid numeric characters
 - UNPKA can be used on EBCDIC
 - Requires translation of destination zone fields
 - Example: OC or TR type operations

PKA and UNPKA Examples



PKA TARGET1, SOURCE1
* TARGET1 -> XL16 '000000000012345C'


TARGET1	DS	XL16	
SOURCE1	DC	CL08 '00012345'	EBCDIC -OR-
SOURCE1	DC	XL08 '3030303132333435'	ASCII

UNPKA TARGET2, TARGET1
* TARGET2 -> XL06 '303132333435'


TARGET2	DS	XL06
---------	----	------

TRTR instruction

- Translate and Test Reversed
 - Opcode X'D0'
 - **TRTR** $D_1(L_1, B_1), D_2(B_2)$ [SS-Format]
- TRTR processes the same as the TRT instruction
 - Including:
 - Setting the condition code
 - Register 1 optionally updated
 - Register 2 optionally updated
 - $D_2(B_2)$ references a 256 byte search table, as usual
 - **Except:**
 - $D_1(L_1, B_1)$ (Arg_1) references the rightmost byte
 - Process proceeds right to left

TRTR example

* Find the last non-blank character in a string

```
XR R2,R2
```

```
TRTR STRING+L'STRING-1(L'STRING),TABLE
```

```
* R2 = X'000000FF'
```

```
* R1 -> Address of the letter 'D'
```

```
STRING DC CL40' HELLO WORLD '
```

↓

```
TABLE DC 256XL1'FF'
```

```
ORG TABLE+C' '
```

```
DC XL1'00'
```

```
ORG ,
```

TRE instruction (1)

- Translate Extended
 - **TRE** R_1, R_2 [RRE-Format]
 - Opcode X'B2A5'
 - R_1 represents an even/odd pair of registers
 - R_1 contains the address of the field to translate
 - R_1+1 contains the length of the field to translate
 - R_2 points to a 256 byte translation table
 - Same use as Arg_2 of the TR instruction
 - General purpose register 0 contains a *test byte*
 - Low order byte (bits 56-63)
 - Remainder of register is ignored
 - Sets a condition code (see next slide)
- Super translate instruction!
 - Not limited to 256 bytes
 - Operates on implementation defined section at a time
 - Includes required termination (test) byte

TRE instruction (2)

- Processes bytes left to right (length > 0)
 - Until a condition code (see below) is set, bytes are translated similar to the TR instruction

Event	R_1	R_1+1	Condition Code
All bytes processed (i.e. Bytes R_1+1)	R_1 points to the end of the string +1	R_1+1 is set to 0	0
Test byte (i.e. low order byte of GPR 0) matched in source	R_1 points to the location of the matched test byte from (GPR 0).	R_1+1 contains the residual length from the location of the test byte match	1
CPU-determined number of bytes processed	The CPU-determined number of bytes is added to R_1 .	The CPU-determined number of bytes is subtracted from R_1+1	3

TRE example



```
*          Translate ASCII to EBCDIC

          XR          R0,R0          Test byte = x'00'

          LA          R2,FIELD       R2 -> Field

          LHI         R3,L'FIELD     R3 = Length

          LARL        R4,ATETAB     R4 -> Translate Table

LOOP     DS          0H             Over all Bytes . . .

          TRE          R2,R4         Translate

          JO          LOOP          More to go (CC = 3)?

          JZ          DONE          Done (CC = 0)?

          LA          R2,1(,R2)     Bump past null (CC=1)

          JCT         R3,LOOP       Any left ?, continue

DONE     DS          0H             Process complete

          . . .

FIELD   DS          XL1000         String to convert
ATETAB  DC          0XL256         Translation table
        DC          XL16'00010203372D2E2F1605250B0C0D0E0F'
        . . .                     Remainder of table
```

Special translation instructions (1)



- Four special translation instructions:
 - **TROO** – Translate One to One
 - Opcode x ' B993 '
 - **TROT** – Translate One to Two
 - Opcode x ' B992 '
 - **TRTO** – Translate Two to One
 - Opcode x ' B991 '
 - **TRTT** – Translate Two to Two
 - Opcode x ' B990 '

Special translation functions (2)

- **Trxx** R_1, R_2 [RRE-Format]
 - R_1 represents an even/odd pair of registers
 - R_1 has the address of the destination
 - R_1+1 has the length of the source
 - R_2 is a single register
 - R_2 has the address of the source
 - General Purpose Register 0
 - Contains test character(s) (similar to the TRE instruction)
 - Bits 48 to 63 (**TRTO**, **TRTT**) or bits 56 to 63 (**TROO**, **TROT**)
 - General Purpose Register 1
 - Contains the address of a translation table
 - Must be on a doubleword (originally, page) boundary

Special translation instructions (3)

- Process similar to TRE
 - Checking test character(s)
 - Setting condition code
- Except:
 - Different source and destination
 - Test character's size
 - Table size and alignment (* ETF-2 relaxes restrictions on page alignment; see slide 26)

Instruction	Source	Destination	Test Character(s)	Table
TROO	One Byte	One Byte	One Byte (bits 56 to 63)	256 bytes (doubleword boundary)
TROT	One Byte	Two Bytes	One Byte (bits 56 to 63)	512 bytes (doubleword boundary)
TRTO	Two Bytes	One Byte	Two Bytes (bits 48 to 63)	64K bytes (page boundary)*
TRTT	Two Bytes	Two Bytes	Two Bytes (bits 48 to 63)	128K bytes (page boundary) *

TROO example



```

*           Translate ASCII to EBCDIC using TROO (Field2 <- Field1)

XR          R0,R0                Test byte = x'00'
LARL       R1,ATETAB             R1 -> Translate Table

LA          R2,FIELD2            R2 -> Field (Destination)
LHI        R3,L'FIELD1          R3 = Length (Source)
LA          R4,FIELD1            R4 -> Field (Source)

LOOP       DS          0H        Overall Bytes . . .
          TROO        R2,R4      Translate
          JO          LOOP       More to go (CC = 3)?
          JZ          DONE       Done (CC = 0)?
          MVC         0(1,R2),0(R4) Copy source to destination
          LA          R2,1(,R2)   Bump past null, dest. (CC=1)
          LA          R4,1(,R4)   Bump past null, source
          JCT         R3,LOOP     Any left ?, continue

DONE       DS          0H        Process complete

          . . .

FIELD1     DS          XL1000     Source String to convert
FIELD2     DS          XL1000     Destination String
          DC          0D'0'       Align
ATETAB     DC          0XL256     Translation table
          DC          XL16'00010203372D2E2F1605250B0C0D0E0F'
          . . .
          Remainder of table
    
```

Special translation instructions (ETF-2 Enhancement Facility)



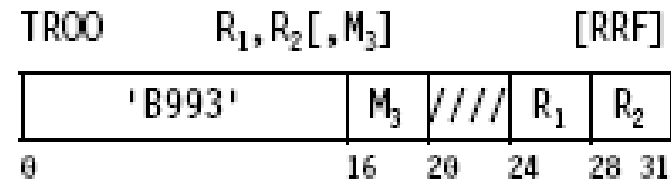
- TROO, TROT, TRTO and TRTT optional processing

- **Trxx R₁, R₂, [M₃]**

- R₁ and R₂ same as before

- M₃ optional 4-bit mask

- Currently bits 0, 1 and 2 must be zero
- Bit 3 of mask (Test Character-Comparison Control)
 - 0 – Operate as before
 - 1 – **Do not perform character comparison**
 - Only translation is performed
 - M₃ Ignored if ETF-2 Enhancement Facility not installed



- Translation-table alignment restriction relaxed:

- TROO and TROT: as before, are on a doubleword boundary

- TRTO and TRTT: new boundary conditions

- No ETF-2: 4K boundary
- With ETF-2: doubleword boundary

Translate and Test Extended Facility (zSeries 10.0 Enhancement)



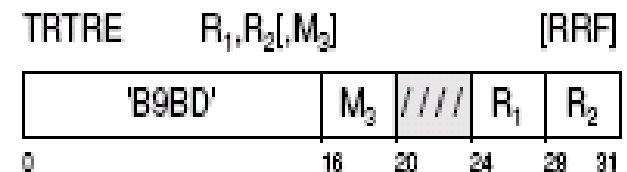
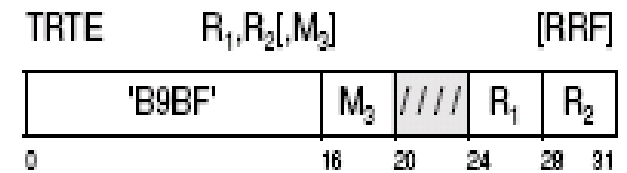
- Instructions

- **TRTE** $R_1, R_2 [, M_3]$

- Translate and Test Extended
 - Enhanced TRT

- **TRTRE** $R_1, R_2 [, M_3]$

- Translate and Test Reverse Extended
 - Enhanced TRTR



- Enhancements:

- Removes 256 byte length limit: uses length in R_1+1
 - Removes GPR R2 restriction: uses R_2
 - Optional Modifier (M_3) controls interpretation
 - Size and value of translate scan entity - (addressed by R_1)
 - Size of translate table - (addressed by GPR R1)