Assembler BootCamp Plus: Instructions Everyone Can Use

(Created by)

John Dravnieks, IBM Australia
(dravo@au1.ibm.com)

(Presented by Dan Greiner and John Ehrman)

Maps by Geoscience Australia

SHARE 115, Boston MA

August 5, 2010

Agenda

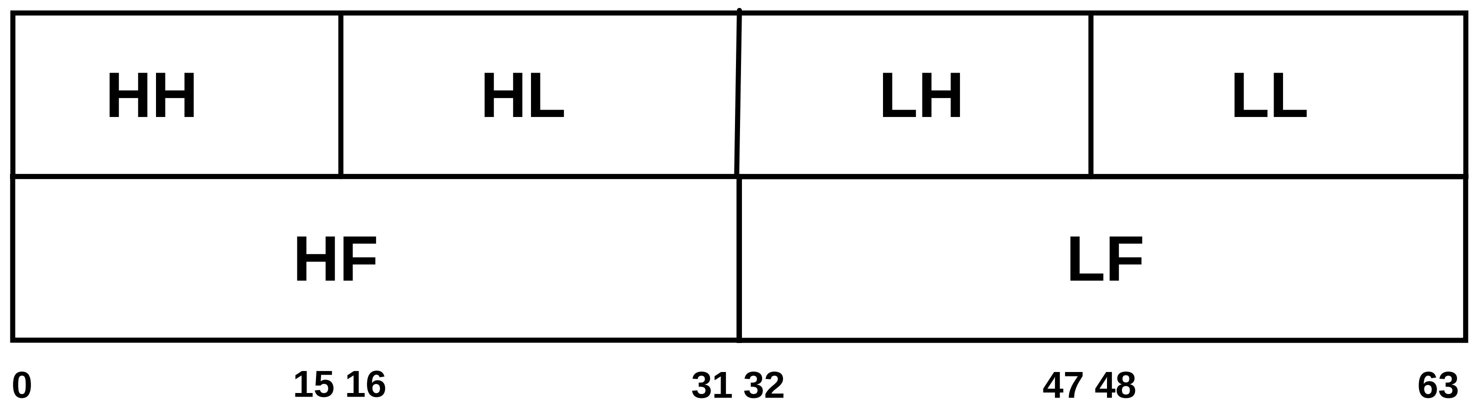
- Bit shifting
- Single byte operands
- Halfword operands
- Multiple byte operands
- Variable length operands
- Character translation

Definitions

- Characters used in instruction mnemonics
  - **G** - **G**rande - 64-bit operand
  - **F** - **F**ullword - 32-bit operand
  - **H** - **H**alfword - 16-bit operand
  - Single byte operands
    - **B** - **B**yte (signed 8 bit value)
    - **C** - **C**haracter
  - **L** - **L**ogical - unsigned, or Load and clear
  - **Y** - 20-bit displacement

Definitions

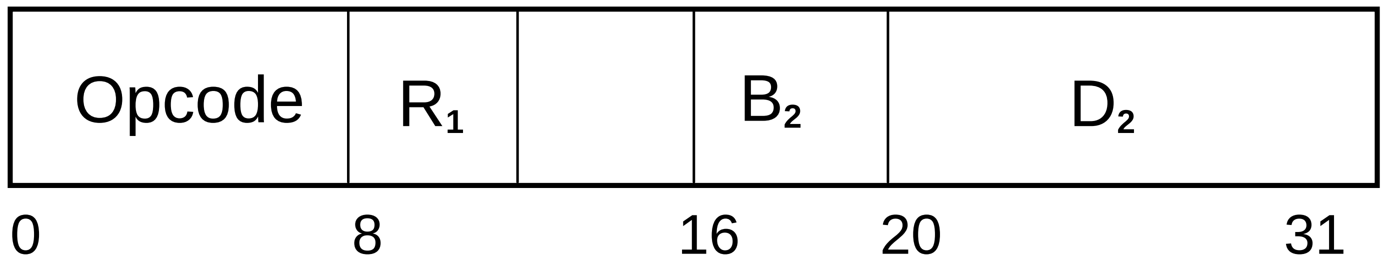
- Parts of a 64-bit register



- H=High, L= Low, F=Fullword

Bit shifting

- RS-type format instructions



- R_1 - Source **and** target
- 2nd operand address
  - **NO** storage reference
  - Last 6 bits used as shift amount

Bit shifting *(continued)*

- Two directions, two types, and two sizes
  - **L**eft or **R**ight
  - **L**ogical or **A**rithmetic
  - Single or **D**ouble register
- 8 mnemonics - **S**hift ...

SLA	**SLDA**
SLL	**SLDL**
SRA	**SRDA**
SRL	**SRDL**

Bit shifting *(continued)*

- 64-bit register instructions
  - **NO** 64-bit-register-pair (128-bit) shifts
  - Single-length: **`SLAG, SRAG, SLLG, SRLG`**

- Separate source (R_3) and target (R_1) registers

- Example:

  $\mathtt{SLAG\ R_1,R_3,D_2(B_2)}$

  - Shifted contents of R_3 goes into R_1

Bit shifting *(continued)*

- Arithmetic shifts:
  - Sign bit not modified
  - Right shifts copy sign bit
  - Left shifts may overflow
  - Condition code set
- Logical shifts:
  - No sign bit
  - Always inserts 0's
  - Condition code not changed

Bit shifting: example 1

- `SRA   5,16`
  - Object code `X'8A50  0010'`
  - c(r5) before `X'8001  0000'` (sign is propagated)
  - c(r5) after `X'FFFF  8001'`
  - Condition code 1 set (result < 0)
- `SRA   5,7`
  - Object code `X'8A50  0007'`
  - c(r5) after `X'FE00  0200'`
- `SRA   5,20`
  - c(r5) after `X'FFFF  F800'`

Bit shifting: example 2

- `SRL 5,16`
  - Object code `X'8850 0010'`
  - c(r5) before `X'8001 FFFF'`
  - **c**(r5) after `X'0000 8001'` **(sign not propagated)**
- `SRL 5,7`
  - Object code `X'8850 0007'`
  - c(r5) after `X'0100 03FF'` `(3 = 0011)`
- `SRL 5,20`
  - c(r5) after `X'0000 0800'`

Bit shifting: example 3

- `SLA   5,16`
  - Object code `X'8B50 0010'`
  - c(r5) before `X'0000 8001'`
  - c(r5) after `X'0001 0000'`
  - Condition code 3 set (Overflow)
- `SLA   5,7`
  - Object code `X'8B50 0007'`
  - c(r5) after `X'0040 0020'` **(CC2, no overflow)**
- `SLA   5,30`
  - c(r5) after `X'4000 0000'` **(overflow)**

Bit shifting: example 4

- `SLL 5,16`
  - Object code `X'8950 0010'`
  - c(r5) before `X'0000 8001'`
  - c(r5) after `X'8001 0000'`
- `SLL 5,7`
  - Object code `X'8950 0007'`
  - c(r5) after `X'0040 0020'`
  - `SLL 5,30`
  - c(r5) after `X'4000 0000'`

Bit shifting *(continued)*

- Rotate Left Single Logical
  - **`RLL(G)   R`**$_1$**`,R`**$_3$**`,D`**$_2$**`(B`**$_2$**`)`**
    - Separate target (R_1) and source (R_3) registers
  - Example: **`RLL    7,8,12(0)`**
  - Before: c(R7)=**`X'????????'`**, c(R8)=**`X'FEDC0000'`**
  - After: c(R7)=**`X'C0000FED'`**, c(R8)=**`X'FEDC0000'`**

Bit shifting: uses

- Arithmetic Operations
  - Fast multiplication or division by a power of 2
  - Hashing algorithms

- Masking
  - In conjunction with Boolean operations
    - Exclusive OR (XOR), OR, AND
  - Extracting data
    - Merged or compressed data

- Encryption

Single byte operands

- **I**nsert **C**haracter
  - `IC    R1,D2(X2,B2)` — **IC** $R_1,D_2(X_2,B_2)$
  - Copies a single byte from storage into low order byte of R_1
  - **Note:** rest of R_1 register unchanged

- **ST**ore **C**haracter
  - **STC** $R_1,D_2(X_2,B_2)$
  - Copies the low order byte of R_1 into storage

Single byte operands: example 1

- `IC   7,0(0,11)`
  - Object text **`X'4370 B000'`**
- R11 points to storage byte containing **`X'A5'`**

- c(R7) before **`X'1234 5678'`**
- c(R7) after **`X'1234 56A5'`**
  - Remainder of register R7 is unchanged

- Condition code is unchanged

Single byte operands

- **L**oad **L**ogical **C**haracter
  - **`LL(G)CR  `**$\mathbf{R_1,R_2}$
  - **`LL(G)C   `**$\mathbf{R_1,D_2(X_2,B_2)}$
  - Clears the register and copies a byte from register or storage into low order byte of R_1

- **L**oad **B**yte
  - **`L(G)BR   `**$\mathbf{R_1,R_2}$
  - **`L(G)B    `**$\mathbf{R_1,D_2(X_2,B_2)}$
  - Single byte from register or storage is sign extended and updates the *entire* register

Single byte operands: example 2

- `LLC 7,0(0,11)` Load Logical Character
  - ► Object text `X'E370 B000 0094'`
- R11 points to storage byte containing `X'A5'`

- c(R7) before `X'1234 5678'`
- c(R7) after `X'0000 00A5'`
  - ► Remainder of register R7 is zeroed

- Condition code is unchanged

Single byte operands: example 3

- `LB 7,0(0,11)` Load Byte
  - Object text **`X'E370 B000 0076'`**
- R11 points to storage byte containing **`X'A5'`**

- c(R7) before **`X'1234 5678'`**
- c(R7) after **`X'FFFF FFA5'`**
  - Leftmost bit of **`X'A5'`** extended to left

- Condition code is unchanged

Single byte operands: uses

- Translation example (we'll use it again):

  -
    ```
      ...
      IC    R2,0(,R3)           Get next character
      N     R2,=X'0000000F'     Keep low-order hex digit
      IC    R2,TABLE(R2)        Use c(R2) as index
      STC   R2,0(,R3)           Store "translated" digit
      ...
    ```

  -
    ```
    TABLE   DC    C'0123456789ABCDEF'
    ```

- The low-order hex digit of each byte referenced by R3 is replaced by its character representation

Halfword (two byte) operands

- RX instructions
  - Mnemonic **$R_1,D_2(X_2,B_2)$**
- Operand 1 is entire R_1 register
  - **STH** ignores high order 16 bits of R_1, stores only rightmost 16 bits
- Operand 2
  - Halfword in storage
  - Signed value - **LH** expands to fullword with sign extension

Halfword (two byte) operands
(continued)

Add **H**alfword	**AH**
Compare **H**alfword	**CH**
Load **H**alfword	**LH**
Multiply **H**alfword	**MH**
STore **H**alfword	**STH**
Subtract **H**alfword	**SH**

Halfword (two byte) operands *(continued)*

- Halfword immediate format
  - **`Mnemonic`** $\mathbf{R_1, I_2}$

    where $\mathbf{I_2}$ is a signed 16-bit field in the instruction

Add **H**alfword **I**mmediate	**AHI**
Compare **H**alfword **I**mmediate	**CHI**
Load **H**alfword **I**mmediate	**LHI**
Multiply **H**alfword **I**mmediate	**MHI**

Halfword (two byte) operands *(continued)*

- Halfword-immediate operands for 64-bit registers:
  - ►**AGHI, CGHI, LGHI, MGHI**
  - ►**LGH(R)**
- Long displacement facility (instructions with signed 20-bit displacement)
  - ►**AHY, CHY, LHY, STHY, SHY**

Halfword (two byte) operands *(continued)*

- Register-to-register form: **L(G)HR**
- Load Logical form: **LL(G)HR, LL(G)H**
- Load Logical Immediate form: **LLIxx**
- Insert Immediate form: **IIxx**
- Where **xx - HH, LH, HL, LL** (See slide 4)

Halfword operands: example 1

- `LH  0,0(0,12)`
  - Object text `X'4800 C000'`
- R12 points to storage containing `X'B1A4'`
- c(R0) before `X'FEDC BA98'`
- c(R0) after `X'FFFF B1A4'`
  - High-order bit of `X'B1A4'` extended to left
- Condition code is unchanged

Halfword operands: example 2

- `CH  10,0(0,11)`
  - Object text **`X'49A0 B000'`**
- R11 points to storage containing **`X'B1A4'`**

- If c(R10) = **`X'FFFF  B1A4'`**
  - Condition code set to 0 (equal)
  - R10 unchanged
- If c(R10) = **`X'0000  B1A4'`**
  - Condition code set to 2 (greater)

Halfword operands: example 3

- `   CH   10,0(0,11)`
  - Object text **`X'49A0 B000'`**
- R11 points to storage containing **`X'B1A4'`**
- If c(R10) = **`X'FFFF  A5A5'`**

- Resulting Condition Code ?
- Is R10 unchanged?

Halfword operands: example 4

- **`LLILH   0,X'A5A5'`**
  - **L**oad **L**ogical **I**mmediate **L**ow **H**igh
  - Object text **`X'A50E A5A5'`**

- c(R0) before **`X'FEDC BA98'`**
- c(R0) after **`X'A5A5 0000'`**
  - Remainder of target register is zeroed

- Condition code is unchanged

Halfword operands: example 5

- `IILH    0,X'A5D6'`
  - **I**nsert **I**mmediate **L**ow **H**igh
  - Object text `X'A502 A5D6'`
- c(R0) before `X'FEDC BA98'`
- c(R0) after `X'A5D6 BA98'`
  - Remainder of target register is unchanged
- Condition code is unchanged

Halfword operands: uses

- Record lengths (DCBLRECL)
  - V format records: RDWs, BDWs
- Database records
- Small integers

Multiple byte operands

- **I**nsert **C**haracters under **M**ask
  - **`ICM  R`$_1$`,Mask,D`$_2$`(B`$_2$`)`**
  - Copies 0 to 4 bytes from storage into mask-selected bytes of R_1
  - Condition code set
  - **Note:** Unselected bytes unchanged

Multiple byte operands *(continued)*

- Mask operand is a 4 bit field
  - Bits correspond one to one with bytes of register
  - **`B'1001'`** refers to the first and last byte
- Storage bytes are contiguous
  - **`  ICM  2,B'1010',=X'12345678'`**
  - c(R2) = **`X'12??34??'`**

Multiple byte operands *(continued)*

- **C**ompare **L**ogical Characters under **M**ask
  - **CLM R_1,Mask,D_2(B_2)**
  - Compares 0 to 4 contiguous bytes from storage with mask-selected bytes of R_1
  - Condition code is set

- **ST**ore **C**haracters under **M**ask
  - **STCM R_1,Mask,D_2(B_2)**
  - Stores 0 to 4 bytes from selected bytes of R_1 into contiguous storage bytes

Multiple byte operands *(continued)*

- z/Architecture instructions:
  - **CLMY, CLMH**
  - **ICMY, ICMH**
  - **STCMY, STCMH**
- **H** = **H**igh-order 32 bits of 64-bit register
- Long-displacement format (RSY)

Multiple byte operands: uses

- `STCM    R`$_1$`,B'0111',D`$_2$`(R`$_2$`)`

  - Stores low-order 24 bits of R$_1$ into contiguous storage bytes

  - Historically important use:

```
–         STCM  R5,B'0111',Label+1
–Label    DC    X'bits',AL3(address)
```

    - DCB address fields
    - CCW address field

Multiple byte operands: uses *(continued)*

- ICM with mask **`B'0001'`**
  - Same as IC, but condition code is set

- ICM with mask **`B'1111'`**
  - Same as Load, but condition code is set

```
 ►  ICM   5,B'1111',24(8)   is equivalent to:
 ►  L     5,24(,8)          this
    LTR   5,5               plus this
```

  - **NO** index register with ICM

Fullword operands

- z/Architecture with extended immediate facility
  - **L**oad and **T**est - **`LT`** (like **`L + LTR`**)
  - 32-bit **F**ullword **`I`**mmediate operands:
    - Arithmetic: **`AFI, ALFI, SLFI`**
    - Logical AND, XOR, OR: **`NIHF, NILF, XIHF, XILF, OIHF, OILF`**
    - Compare: **`CFI, CLFI`**
    - Load immediate: **`LGFI, LLIHF, LLILF`**
    - Insert immediate: **`IIHF, IILF`**

Variable number of operand bytes

- Q: How would we store HLASM symbols, from 1 to 63 bytes long?
- A1: Update MVC instruction in storage?
  - Reentrancy violation
  - Difficult to debug
  - Data / Instruction cache conflicts?
- A2: Use **`IC`** and **`STC`** in a loop?
  - Slow
- A3: Use **EX**ecute instruction!

EXecute instruction

- `EX   R`$_1$`,D`$_2$`(X`$_2$`,B`$_2$`)`
- Operand 2 - Address of target instruction
- If R_1 is not general register 0, then low order byte is ORed internally with the **second** byte of the target instruction
- The target instruction is then performed
  - The target instruction in memory is unchanged!

EXecute instruction *(continued)*

- Three important points
  - Operands 1 and 2 are not modified
  - The operation is a logical OR
  - When EXecuting variable-length instructions, lengths in object text are one less than actual length
- An example follows

EXecute instruction example

- `         EX    R4,MOVEIT`

- `MOVEIT    MVC TARGET(0),SOURCE`
  - Object text **`X'D200  bddd  bddd'`**

- c(R4) = **`X'1234 5602'`**

- Effective object text **`X'D202 bddd  bddd'`**

- So three (3) bytes are moved

EXecute instruction: lengths

- R4 in that example holds *machine* length

- If R4 holds *actual* length, then how do we make R4 the machine length (one less)?
  - Any one of these:
  - **`S    R4,=F'1' (or SH R4,=H'1')`** (?)
  - **`BCTR R4,0`**
  - **`LA   R4,255(,R4)`**
  - **`AHI  R4,-1      (Recommended!)`**

EXecute instruction: uses

- Usually, the target instruction is SS-type, like **`MVC, CLC, TR`** or **`TRT`**

- **`NOP`** (or BC 0) can be **`EX`**ecuted
  - OR in mask of **`X'F0'`** to skip code

- Example:
- **`EX 0,Target_SVC`**
  - Allows shared code (Test and Production) to use different SVCs

Variable number of operand bytes - Take 2

- Q: How would we store character strings from 1 to 567 bytes long?
- A1: Update instruction in storage **(Bad!)**
  - Won't work anyway: max length is 256
- A2: Use **`IC`** and **`STC`** in a loop?
  - Even slower
- A3: Use **EX**ecute instruction? (Not bad...)
  - Loop moving 256 byte chunks and then an EXecuted move at the end (used in old days)
- A4: Use Move Long!

Move Long instruction

- **`MVCL      R`$_1$`,R`$_2$**
  - **`MVCL 4,6`** - object text **`X'0E46'`**

- Operands designate even-odd register pairs:
  - Even register: operand address
  - Odd register (even+1): operand length
    - Source length register has pad character in high order byte
    - Maximum length is 16MB (24 remaining bits of the odd registers)

Move Long instruction *(continued)*

- All 4 registers may be modified
- Sets condition code
- R0 (implying the pair R0 and R1) is valid
  - Yes, R0 *can* contain an address!
- Clear a block of storage:

```
►     LM    0,3,=A(Block,L'Block,0,0)
►     MVCL  0,2    X'00' Pad char in R3
```

Compare Logical Long instruction

- **`CLCL      R`$_1$`,R`$_2$**
  - **`CLCL 4,6`** - object text **`X'0F46'`**

- Same register setup as MVCL

- All 4 registers may be modified - data in storage is NOT modified
- Shorter operand padded with pad character
- Condition code is set

CLCL example

- Example of CLCL usage

```
LM      2,3,=A(String1,L'String1)
```

Target addr, length

```
LM      0,1,=A(String2,L'String2)
```

Source addr, length

```
ICM     1,B'1000',=C' '        Pad byte
CLCL    2,0
BE      Equal_strings
```

Extended Move and Compare Long

- Move Long Extended (**`MVCLE`**)
- Move Long Unicode (**`MVCLU`**)
- Compare Logical Long Extended (**`CLCLE`**)
- Compare Logical Long Unicode (**`CLCLU`**)

  - Lengths can be greater than 16MB
  - Pad character formed from 2nd operand
  - Unicode: 2 bytes per step
  - CC set to 3 if operation is incomplete

Extended Move and Compare Long - examples

```
Compare  CLCLE 2,0,X'40' blank pad
         BO    Compare   CC3 test
         BE    Equal_strings


CompUni  CLCLU 2,0,X'020'
         BO    CompUni   CC3 test
         BE    Equal_strings
```

Move with Optional Specifications

- **`MVCOS`** $\mathbf{D_1(B_1),D_2(B_2),R_3}$
  - Set GPR0 to zero
  - Set R3 operand to **TRUE** length

  - Moves 0 - 4096 bytes
    - If true length greater than 4096, then 4096 bytes moved and condition code 3 is set
    - Otherwise, true length bytes moved and condition code 0 is set

Translation

- Q: How to ensure that character data is in upper case?
- A1: Use the IC/STC code earlier (slide 20) with a new table
- A2: Use TRanslate instruction!

TRanslate instruction

- **TR** **$D_1(L_1,B_1),D_2(B_2)$** SS format
- Operand 1 is source **and** target
- Operand 2 is address of translate table
  - Usually 256 bytes - depends on data

```
         TR  STR,Table
STR      DC  C'Hello, World!'
Table    DC  C'......'
```
(See notes)

TRanslate instruction *(continued)*

- Each byte in operand 1 is used to index into operand 2; that byte from table replaces the source byte

-
  ```
  TR    STR,TABLE
  ```
  - Single instruction replaces previous six instruction loop (see note on slide 20)

TRanslate instruction - example

- Translate hex data to printable characters

```
►      UNPK STRING(L'STRING+1),HEXDATA(L'HEXDATA+1)
►*     Get data into zoned format
►      LA R5,L'STRING-1      Load machine length
►      EX   R5,TR_INST       Perform translation
►      ...
►TR_INST TR STRING(0),TABLE Executed TRANSLATE
►      ORG*-240              Position label
►TABLE DS 0X                 Start of 256 byte table
►      ORG*+240              Skip to actual data
►      DC    C'0123456789ABCDEF'
```

Related instructions

- Translate and Test
  - **TRT** $\mathbf{D_1(L_1,B_1),D_2(B_2)}$ Left to right
  - **TRTR** $\mathbf{D_1(L_1,B_1),D_2(B_2)}$ Right to left
- Operands not modified
- Table - operand 1 byte used as index
  - If table byte is zero, scan continues
  - If non zero, scan stops
    - GR1: Address of operand 1 byte
    - GR2: Test-table byte

Related instructions

- Translate Extended
  - **`TRE R1,R2`**
    - First operand address in register R_1
    - First operand length in register R_1+1
    - Translate table address in register R_2
  - Test byte in GR0
    - Translation stops if it matches source byte
    - Registers updated

TRT instruction - example

- Scan for ASCII (**`X'20'`**) or EBCDIC (**`X'40'`**) blanks

```
►    SR     R2,R2                     Clear R2
►    LA     R1,STRING+L'STRING-1 Set R1 to last byte
►    LA     R5,L'STRING-1        Load machine length
►    EX     R5,TRT_INST          Perform scan
►    JZ     No_Blanks            Nothing found (CC 0)
►    CHI    R2,X'20'             ASCII blank?
►         ...
►TRT_INST TRT STRING(0),TABLE Executed TRT
►TABLE   DC    256X'00'          Define 256 byte table
►    ORG    TABLE+X'20'          Move to offset X'20'
►    DC     X'20'                Set non zero
►    ORG    TABLE+X'40'          Move to offset X'40'
►    DC     X'40'                Set non zero
►    ORG,                        Skip to end of TABLE
```

Summary

Many useful instructions!

- Bit shifting
- Single byte operands
- Halfword operands
- Multiple byte operands
- Variable length operands
- Character translation