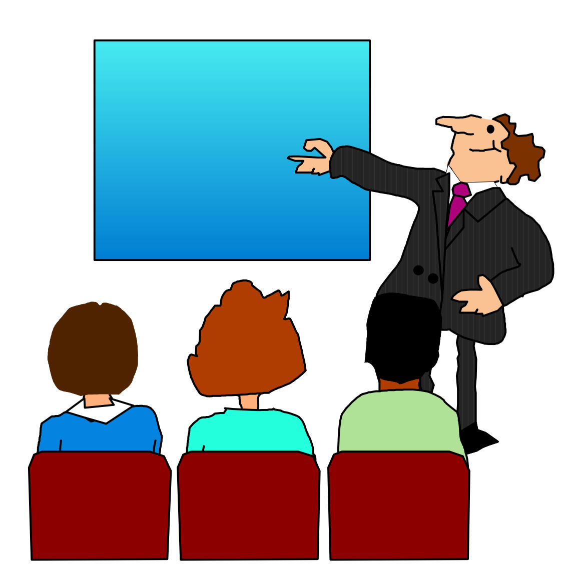
Assembler Language "Boot Camp" Part 3x - Implicit Addresses and USING

SHARE in New York City

August 15 - 20, 2004

Session 8188



Introduction

- Who are we?
  - John Dravnieks, IBM Australia
  - John Ehrman, IBM Silicon Valley Lab
  - Michael Stack, Department of Computer Science, Northern Illinois University

Introduction

- Who are you?
  - An applications programmer who needs to write something in S/390 assembler?
  - An applications programmer who wants to understand S/390 architecture so as to better understand how HLL programs work?
  - A manager who needs to have a general understanding of assembler?
- Our goal is to provide for professionals an introduction to the S/390 assembly language

Introduction

- These sessions are based on notes from a course in assembler language at Northern Illinois University

- The notes are in turn based on the textbook, Assembler Language with ASSIST and ASSIST/I by Ross A Overbeek and W E Singletary, Fourth Edition, published by Macmillan

Introduction

- The original ASSIST (Assembler System for Student Instruction and Systems Teaching) was written by John Mashey at Penn State University

- ASSIST/I, the PC version of ASSIST, was written by Bob Baker, Terry Disz and John McCharen at Northern Illinois University

Introduction

- Both ASSIST and ASSIST/I are in the public domain, and are compatible with the System/370 architecture of about 1975 (fine for beginners)

- Both ASSIST and ASSIST/I are available at http://www.cs.niu.edu/~mstack/assist

Introduction

- Other materials described in this session can be found at the same site, at http://www.cs.niu.edu/~mstack/share
- Please keep in mind that ASSIST and ASSIST/I are not supported by Penn State, NIU, or any of us

Introduction

- Other references used in the course at NIU:
  - Principles of Operation (PoO)
  - System/370 Reference Summary
  - High Level Assembler Language Reference

- Access to PoO and HLASM Ref is normally online at the IBM publications web site

- Students use the S/370 "green card" booklet all the time, including during examinations (SA22-7209)

Our Agenda for the Week

- Session 8181: Numbers and Basic Arithmetic
- Session 8182: Instructions and Addressing
- Session 8183: Assembly and Execution; Branching
- Session 8188: Implicit Addresses and USING ["Extra Credit" Session]

Our Agenda for the Week

- Session 8184: Arithmetic; Program Structures
- Session 8185: Decimal and Logical Instructions
- Session 8186: Assembler Lab Using ASSIST/I

Today's Agenda

- A Computer Program
- A Detailed Look at the USING Instruction
- The Problem: To Use Implicit Addresses Everywhere!
- The Solution: We Fool the Assembler
- The Last Step: We Eliminate Overlapping USING Ranges

The Agenda (cont'd)

- The Final Last Step: We Stop Wasting Space
- Conclusions

Overview

- The USING instruction is usually the most difficult instruction for assembler students to learn and understand
- USING makes assembler programs simpler and much more maintainable
- The approach we will use is "pragmatic," in that we will decide which USING and DROP instructions to use based on the object code - the "bits" - to be generated

Overview - and Caution

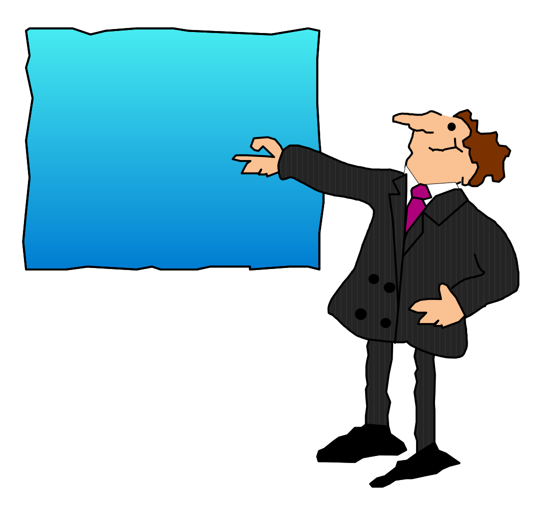
- Most discussions of assembler language programming begin with assembler language instructions, then proceed to show what bits they generate

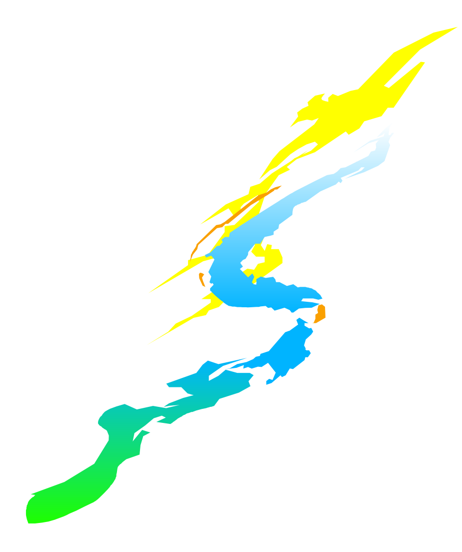
- We will first look at an executable program as the computer sees it, then explore ways to create, or "generate," that program

- This will help us understand how to use the USING instruction

A Computer Program

In Which We Pretend That Computers Can Count To One





A Program as the Computer Sees It, in Binary (Slightly Abridged)

```
1001000011101100110100000000110001000000111000000111000000011000
0101000011010000111000000000001000100000001110000011010000000001000
0001100011011110000000101110000000010001111111000011000000001001100
1111111111111111111111111111111111111111111111111111111111111 ...
0100000010001000011000000011010000100010111100000110000001000010000
0100000010001000011000000010111100010001011110000011000000100010000
0101100011010000110100000000010010011000111011001101000000001100
0001101111111111000001111111111001010101010101010101010101 ...
0101010101010101010101010101010100011011000000000100000110100000
0000000000000011000110001011000111010010000101011011000000000000
1101000100101100010100000000000000101100000000110000100000110110000
1011000000011100010001101010000011000000100011000000000111111110
0100000001000000010000000100000001000000010000000100000001000000
0100000001000000010000000100000001000000010000000100000001000000
010000000100000001000000010000000100000001000000
```

A Program as the Computer Sees It, Organized into Bytes

```
10010000 11101100 11010000 00001100 01000001 11100000 11110000 00011000
01010000 11010000 11100000 00000100 01000000 11100000 11010000 00001000
00011000 11011110 00000101 11000000 01000111 11110000 11000000 01001100
11111111 11111111 11111111 11111111 11111111 11111111 11111111 1111 ...
01000001 00010000 11000000 01101000 01000101 11100000 11000001 00010000
01000001 00010000 11000000 10111100 01000101 11100000 11000001 00010000
01011000 11010000 11010000 00000100 10011000 11101100 11010000 00001100
00011011 11111111 00000111 11111110 01010101 01010101 01010101 0101 ...
01010101 01010101 01010101 01010101 00011011 00000000 01000001 10100000
00000000 00000011 00011000 10110001 11010010 00010101 10110000 00000000
11010001 00101100 01010000 00000000 10110000 00011000 01000001 10110000
10110000 00011100 01000110 10100000 11000001 00011000 00000111 11111110
01000000 01000000 01000000 01000000 01000000 01000000 01000000 01000000
01000000 01000000 01000000 01000000 01000000 01000000 01000000 01000000
01000000 01000000 01000000 01000000 01000000 010000000
```

A Program as the Computer Sees It, Organized as Instructions (Pt 1)

```
10010000 11101100 11010000 00001100
01000001 11100000 11110000 00011000
01010000 11010000 11100000 00000100
01000000 11100000 11010000 00001000
00011000 11011110
00000101 11000000
01000111 11110000 11000000 01001100
11111111 11111111 11111111 1111 ...
01000001 00010000 11000000 01101000
01000101 11100000 11000001 00010000
01000001 00010000 11000000 10111100
01000101 11100000 11000001 00010000
01011000 11010000 11010000 00000100
10011000 11101100 11010000 00001100
00011011 11111111
00000111 11111110
01010101 01010101 01010101 0101 ...
```

A Program as the Computer Sees It, Organized as Instructions (Pt 2)

```
00011011 00000000
01000001 10100000 00000000 00000011
00011000 10110001
11010010 00010101 10110000 00000000 11000001 00101100
01010000 00000000 10110000 00011000
01000001 10110000 10110000 00011100
01000110 10100000 11000001 00011000
00000111 11111110
01000000 01000000 01000000 01000000
01000000 01000000 01000000 01000000
01000000 01000000 01000000 01000000
01000000 01000000 01000000 01000000
01000000 01000000 01000000 01000000
01000000 01000000
```

A Program as the Computer Sees It, Instructions in Hex (Pt 1)

```
90ECD00C
41E0F018
50D0E004
50E0D008
18DE
05C0
47F0C04C
FFFF ...
4110C068
45E0C110
4110C0BC
45E0C110
58D0D004
98ECD00C
1BFF
07FE
5555 ...
```

A Program as the Computer Sees It, Instructions in Hex (Pt 2)

```
1B00
41A00003
18B1
D215B000C12C
5000B018
41B0B01C
46A0C118
07FE
40404040
40404040
40404040
40404040
40404040
4040
```

A Program is Just Bits in Memory

- All of the previous slides show the same program
- The only differences are in the way it is displayed
- Remember that hexadecimal is simply shorthand for binary
- Now we look at how this program might have been written in assembler language

One Way to Write the Program in Assembler Language (Pt 1)

```
90ECD00C      STM   14,12,12(13)      R13 has address of a save area
41E0F018      LA    14,24(,15)        R15 has address of 1st instruction
50D0E004      ST    13,4(,14)         R14 has address of new save area
50E0D008      ST    14,8(,13)
18DE          LR    13,14
05C0          BALR  12,0              R12 has address of next inst
47F0C04C      BC    15,76(,12)
FFFF ...      DC    18F'-1'
4110C068      LA    1,104(,12)
45E0C110      BAL   14,272(,12)
4110C0BC      LA    1,188(,12)
45E0C110      BAL   14,272(,12)
58D0D004      L     13,4(,13)
98ECD00C      LM    14,12,12(13)
1BFF          SR    15,15
07FE          BCR   15,14
5555 ...      DC    (3*28)X'55'
5555 ...      DC    (3*28)X'55'
```

One Way to Write the Program in Assembler Language (Pt 2)

```
1B00          SR    0,0
41A00003      LA    10,3
18B1          LR    11,1                 R11 has address of table
D215B000C12C  MVC   0(22,11),300(12)
5000B018      ST    0,24(,11)
41B0B01C      LA    11,28(,11)
46A0C118      BCT   10,280(,12)
07FE          BCR   15,14
40404040      DC    CL22' '
40404040
40404040
40404040
40404040
4040
```

That's All There Is To It!

- That's it; we've written a program which, when assembled, will generate the bits - the "object code" - we need
- But how hard is it to write such a program, since we have to calculate all the displacements manually?
- And who would ever write in assembler if all displacements had to be calculated that way?

Can't We Make It Easier?

- We can, but we must always remember that

Object code is all that matters!

- That is, we can write an assembler program source any way we wish, as long it generates as its "object code" the set of bits we want for our executable program
- Everything that follows in this presentation depends on observing this rule

A "Brute Force" Assembly Method

- Is there any other way to generate the correct object code for these statements?
- Here are two very odd possibilities for the second instruction, both of which generate the correct object code, **41E0F018**:

```
DC      X'41E0F018'

DC      F'1105260568'
```

A "Brute Force" Assembly Method

- But these are much more difficult than writing **`LA   14,24(,15)`** with an explicit address in operand two
- We will be much better served by letting the assembler do as much of the work as possible
- This means using labels wherever possible
- And this requires appropriate use of USING

A Detailed Look at the USING Instruction

In Which We Grapple With Whether to Tell the Truth

Use Labels for Address Operands

- The most important way we can make writing assembler programs easier is to write each address operand as a label rather than as explicit displacement and base register **`D(B)`** or **`D(,B)`**

- Of course, the assembler must then convert each label to the appropriate displacement and base register

Use Labels for Address Operands

- The key to this conversion is the USING instruction
- Each USING specifies a base address (usually a label) and one or more registers
  - (We will keep things simple and never have more than one register in a USING)
- Let's look at the first part of the program written in a more "user friendly" fashion

First Part of Program, with Labels

```
 1 APROGRAM CSECT
 2 STMINST  STM   14,12,12(13)         Save all registers
 3          USING APROGRAM,15          Tell assembler about temporary base
 4 *
 5 LAINST   LA    14,SAVEAREA          Get address of a new save area
 6          ST    13,4(,14)            Set it to point to old
 7          ST    14,8(,13)             and set old to point to new
 8          LR    13,14                Set address of available save area
 9 *
10          BALR  12,0                 Get address of next instruction
11          DROP  15                   Temp base no longer available
12          USING GOAROUND,12          Tell assem this reg is available
13 GOAROUND B     SAVEAREA+18*4        Go around the save area
14 SAVEAREA DC    18F'-1'
          ...
```

The First Two USINGs

- There are two USING instructions in this brief section of code
- The first, at statement 3, could be thought of as a promise that register 15 will contain the memory address of APROGRAM at the time the program is executed
- This information is used only once, when statement 5 is assembled as **`41E0F018`** (it is ***not*** used in statement 13)

The First Two USINGs

- The second, at statement 12, assumes that register 12 will contain the memory address of the **B** instruction at statement 13 when the program executes
- The **BALR** at statement 10 assures this when it executes (as **05C0**)
- This information will be used often in the program, but only once in the slide, when statement 13 is assembled as **47F0C04C**

How Does USING Work?

- Remember that an assembler source file is just input data to the assembler program
- The assembler maintains a "base register table" of 16 entries, one for each register
- When the assembler encounters (reads) a statement like "USING baseaddress,reg" it locates the B.R. table entry for "reg" and places the location that "baseaddress" refers to in that entry

How Does USING Work?

- Each control section has been assigned an identifying number, and this "section ID" is also recorded in the entry
- When a "DROP reg" statement is encountered, the assembler clears the B.R. table entry for "reg"
- When an implicit address operand (a label) is encountered, the assembler searches the B.R. table for the "best" base register

How Does USING Work?

- Only those B.R. table entries whose Section ID matches the CSECT containing the label being processed are checked for "best"
- Here is what a portion of the Base Register Table looks like after the USING at statement 12:

...	Section ID	Base Addr
Reg 11		
Reg 12	1	000014
Reg 13		
Reg 14		
Reg 15		

How Does USING Work?

- When a choice of base registers is possible, the assembler will always choose the base register which gives the smallest displacement
- In case more than one register gives the smallest displacement, the assembler will always choose the highest numbered register
- These rules are important in the following

Two USING Definitions

- The domain of a USING instruction begins where the instruction appears in the program, and continues to the end of the program, except when
  - A subsequent DROP specifies the register
  - A subsequent USING specifies the register
- The range of a USING instruction is the 4096 bytes beginning at the base address
  - Range does not extend past the end of the CSECT which contains the base address

The Domain of USING

```
 1 APROGRAM CSECT
 2 STMINST  STM   14,12,12(13)
 3          USING APROGRAM,15    <----+
 4 *                                  |
 5 LAINST   LA    14,SAVEAREA         |
 6          ST    13,4(,14)           | Domain of USING
 7          ST    14,8(,13            | at statement 3
 8          LR    13,14               |
 9 *                                  |
10          BALR  12,0                |
11          DROP  15    <-------------+
12          USING GOAROUND,12   <-------+
13 GOAROUND B     SAVEAREA+18*4         |
14 SAVEAREA DC    18F'-1'               | Domain of USING
...                                     | at statement 12
                                        V
                                   (This continues to the end
                                    of the assembly, or to a
                                    DROP or USING for register 12)
```

The Range of USING

```
 1 APROGRAM CSECT   <------------------+
 2 STMINST  STM   14,12,12(13)         |
 3          USING APROGRAM,15          |
 4 *                                   |
 5 LAINST   LA    14,SAVEAREA          |
 6          ST    13,4(,14)            | Range of USING
 7          ST    14,8(,13             | at statement 3
 8          LR    13,14                |
 9 *                                   |
10          BALR  12,0                 |
11          DROP  15                   |
12          USING GOAROUND,12          |
13 GOAROUND B     SAVEAREA+18*4        |  <----+
14 SAVEAREA DC    18F'-1'              |       | Range of USING
...                                    |       | at statement 10
                                       V       V
                                     (Ranges continue for up to 4096
                                      bytes, or to end of the CSECT,
                                      whichever is first)
```

Some Observations

- The STM instruction in statement 2 is not within the **domain** of any USING, so it cannot have an implicit address (a label) for its third operand, which must be given as an explicit address (**D(B)**)

- Also, statement 2 is not within the **range** of the second USING, so that label STMINST cannot be specified as an implicit address on any instruction following the DROP at statement 11

Overlapping Domains and Ranges

- Note that there are overlapping ranges for statements 13 and following
- Instructions may also fall within domains which overlap (remove the DROP at 11 and the domain of the USING at 3 will continue)
- When this occurs, the assembler is faced with choosing among multiple base registers when converting a label to base and displacement

Examination of Statement 5

- Let's examine statement 5 in detail

- We see immediately that
  - It lies within the domain of the USING at 3
  - SAVEAREA is within the range of that USING
  - R15 is therefore an available base register
  - It is the only available register, so is selected

- Since SAVEAREA is 4+4+4+4+2+2+4 = 24 = X'18' bytes into the range, the instruction is assembled as **41E0F018**

Examination of Statement 13

- We see immediately that
  - It lies within the domain of the USING at 12
  - SAVEAREA is within the range of that USING
  - R12 is therefore an available base register
  - It is the only available register, so is selected

- Since SAVEAREA+18*4 is 4+18*4 = 76 = X'4C' bytes into the range, the instruction is assembled as **`47F0`**<u>`C04C`</u>

What If We Remove the "DROP 15" (Statement 11)?

- Then the assembler can choose between registers 12 and 15 when assembling statement 13

- But register 12 will give the smallest displacement, so 12 is selected (instead of 15)

What If We Don't Tell the Truth?

- Consider what would happen if we mistakenly (or otherwise) code

```
3                    USING LAINST,15
```

- Since SAVEAREA is now 4+4+4+2+2+4 = 20 = X'14' bytes into the range, the **LA 14,SAVEAREA** instruction at statement 5 is assembled as **41E0F<u>014</u>** not **41E0F<u>018</u>**

- The effect of this is that the address of the **B** instruction at GOAROUND is placed in R14

What If We Don't Tell the Truth?

- This obviously doesn't give us the object code we want
- But it does show that the assembler is gullible and will do whatever we tell it
- We can use this gullibility to cause the assembler to generate exactly the object code we want

The Problem: To Use Implicit Addresses Everywhere!

Which We Insist On, or Demand "Double Our Money Back"

The Next Part of the Program

- The next part of the program calls a subroutine named INIT, passing in R1 the address of a table to be initialized

- After initializing both tables, the program terminates

- You might note that INIT won't know which table is being passed

The Next Part of the Program

```
             ...
15 *
16          LA    1,TABLE1             Get address of first table
17          BAL   14,INIT              Go process it
18          LA    1,TABLE2             Get address of second table
19          BAL   14,INIT              Go process it
20 *
21          L     13,4(,13)            Restore save area address
22          LM    14,12,12(13)         Restore other registers
23          SR    15,15                Set return code = 0
24          BR    14                   Return to caller
25 *
26 *    Define tables, with each entry containing Name and Age
27 *
28          DS    0F                   Fullword alignment
29 TABLE1   DC    (3*28)X'55'          Simple table with 3 entries
30 TABLE2   DC    (3*28)X'55'          Simple table with 3 entries
31 *
             ...
```

The Problem

- This brings us to the key issue
- We don't want to write INIT with duplicate code for TABLE1 and TABLE2
- But INIT can't tell which table is being passed during a given call
- Further, we haven't labeled each entry in each table

The Problem

- So, we have to write explicit addreses rather than implicit, and this is undesirable

```
              ...
34 INIT       SR    0,0                        Initial age value=0
35            LA    10,3                       Get loop counter (table size)
36            LR    11,1                       Copy passed table address
              ...
38 *
39 LOOP       MVC   0(22,11),BLANKS  *1* Set name to blanks
40            ST    0,24(,11)        *2*  and age to zero
41            LA    11,28(,11)       *3* Get to next entry
              ...
44            BCT   10,LOOP                    Continue until done
              ...
48 BLANKS     DC    CL22' '                    Initial value for name
              ...
```

"We Want Implicit Addresses!"

- Now, recall the object code we want for the three instructions at 39 - 41, as we saw in our early slides showing the program:

```
D215B000C12C  MVC   0(22,11),300(12)
5000B018      ST    0,24(,11)
41B0B01C      LA    11,28(,11)
```

- Note that we've improved on this slightly by being able to write the label BLANKS as the second operand of the MVC, because we have program base register 12 available

"We Want Implicit Addresses!"

- Wouldn't it be much better to write

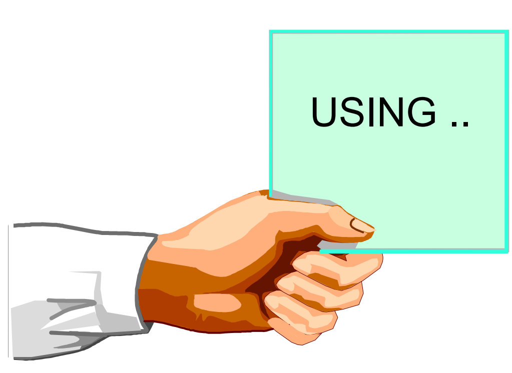
```
               ...
39 LOOP        MVC   NAME,BLANKS *1* Set name to blanks
40             ST    0,AGE       *2*  and age to zero
41             LA    11,NEXT     *3* Get to next entry
               ...
```

  and still generate the same correct object code?

- We will now see how to do just that by "fooling" the assembler

The Solution: We Fool the Assembler

In Which We Make Promises We Have No Intention of Keeping



The Goal

- Our requirement, no matter how it is done, is to use implicit addresses to generate the correct object code

- For instance, if we want to use the label NAME in statement 39, then NAME must have length 22 and there must be a USING which will cause the assembler to generate object code **D215B000C12C**; that is,operand 1 must resolve as displacement 0 and base register 11

The Goal

- And if we are to use AGE at statement 40, it must be on a fullword boundary at displacement 24 from base register 11
- Finally, to use NEXT at 41 means it must be at displacement 28 from register 11
- These requirements must be satisfied in order for the assembler to generate the correct object code

The Solution

- Our solution has two parts
- First, we define fields labeled as desired, in the format we need
- Let's place them just after BLANKS

```
                 ...
47               DS     0F                           Fullword alignment
48 BLANKS        DC     CL22' '                      Initial value for name
                 ...
51 NAME          DS     CL22                         Name field (character)
52 AGE           DS     F                            Age (binary)
53 NEXT          DS     0H                           Next entry
                 ...
```

The Solution

- Then we trick the assembler
- Since we want R11 as the base register, we tell the assembler that, at execution time, R11 will have the address of NAME
- This is clearly a fib, since R11 will actually have the address of an entry in a table
- But the assembler will believe us and will generate the correct object code

The Implementation

- We implement the solution by adding one USING instruction and one DROP to our original LOOP code

- We place the instructions at 39 - 41 between the two

- They are thus within the domain of the added USING (which, incidentally, will overlap the domain of the USING at statement 12)

The Implementation

```
                 ...
34 INIT          SR     0,0                         Initial age value=0
35               LA     10,3                        Get loop counter (table size)
36               LR     11,1                        Copy passed address of table
37               USING NAME,11                      Define base
38 *
39 LOOP          MVC    NAME,BLANKS            *1*  Set name to blanks
40               ST     0,AGE                  *2*   and age to zero
41               LA     11,NEXT                *3*  Get to next entry
42               DROP   11                          No longer needed
43 *
44               BCT    10,LOOP                     Continue until done
                 ...
47               DS     0F
48 BLANKS        DC     CL22' '                     Initial value for name
                 ...
51 NAME          DS     CL22                        Name field (character)
52 AGE           DS     F                           Age (binary)
53 NEXT          DS     0H                          Next entry
                 ...
```

The Implementation

- We have to be careful, though, because this is a situation with overlapping domains and ranges

- In statement 39, the label NAME falls within the range of USINGs at both stmts 12 and 37

- But register 11 yields a displacement (0) much less than that of register 12 (322_{10})

- So the assembler selects R11 as the base

The Implementation

- In statement 39 again, the label BLANKS does **not** fall within the range of the USING at stmt 37 (since the range begins at 51), so the assembler must choose R12 for the base (and 300 for the displacement) for operand 2
- Finally, the length attribute of NAME is 22_{10} so the object code generated is correct
- AGE and NEXT are handled the same way

N.B.: Understanding Required Here

- It is most important to understand that at no time do any of these three instructions reference the storage areas labeled NAME, AGE, and NEXT
- This is because R11 actually has the address of a table entry
- So, for example, the MVC instruction at stmt 39 does **not** alter the data area at label NAME, even though it appears to do so

We're Almost Done

- In fact, if it weren't for a potential problem introduced by overlapping ranges, we could be satisfied with this solution
- Unfortunately, errors are easily introduced, as can be seen if we move BLANKS

```
                ...
47              DS      0F
51 NAME         DS      CL22                 Name field (character)
52 AGE          DS      F                    Age (binary)
53 NEXT         DS      0H                   Next entry
48 BLANKS       DC      CL22' '              Initial value for name
54 *
                ...
```

We're Almost Done

- Now BLANKS lies within the ranges of both USINGs, and the assembler - following its rules - will chose the base register giving the smallest displacement

- This will result in statement 39 being assembled as **`D215B000B01C`**, which is clearly incorrect

- So, our lie has gotten us into trouble, and we need a way out

We're Almost Done

- But please note that moving BLANKS fixed a subtle alignment problem
- We initially placed BLANKS on a fullword boundary, and this forced NAME off a fullword boundary so that the expected two-byte gap before AGE was missing
- This originally caused the ST instruction at 40 to be assembled as **`5000B016`** which is clearly an error (not mentioned earlier)

The Last Step: We Eliminate Overlapping USING Ranges

In Which We Find a New Use for Control Sections

Use CSECT to Isolate Labels

- The simplest way to avoid the overlap problem is to define the labels NAME, AGE, and NEXT so they are outside the range of any USING other than the one at stmt 37

- That will force the assembler to choose R11 as the base

- Recalling that ranges do not extend beyond CSECT boundaries, we can place our labels in a separate CSECT

Use CSECT to Isolate Labels

- Thus, one solution might be

```
                ...
48 BLANKS     DC     CL22' '                  Initial value for name
50 ENTRY      CSECT
51 NAME       DS     CL22                     Name field (character)
52 AGE        DS     F                        Age (binary)
53 NEXT       DS     0H                       Next entry
                ...
```

- Now the labels are defined outside the range of the USING at 12 because they are in a separate CSECT (which is usually placed at the end of the program)

- Note: This also fixes the alignment error

The Final Last Step: We Stop Wasting Space

In Which We Clean Up Our Act

A Last, Small Step

- The storage areas we have defined - NAME, AGE, and NEXT - have attributes we need
- In particular, the displacement of each from NAME (or ENTRY), and the length attribute of NAME are used
- But the storage defined by these DS instructions at statements 51 - 53 is never accessed

A Last, Small Step

- Since the storage is never accessed, it is wasted
- The amount of storage in this case is minimal, but in other cases it might be significant
- To avoid wasted space in this situation, we can replace the CSECT instruction with DSECT (defines a Dummy SECTion)

A Last, Small Step

- Our code would then be

```
                 ...
48 BLANKS    DC     CL22' '                     Initial value for name
49 *
50 ENTRY     DSECT
51 NAME      DS     CL22                        Name field (character)
52 AGE       DS     F                           Age (binary)
53 NEXT      DS     0H                          Next entry
                 ...
```

- The labels within a DSECT carry the same attributes as those in a CSECT, but the assembler allocates no storage for a DSECT

- "Waste not, want not"

Conclusions

In Which We Summarize What Was Covered, and Suggest a Problem

In Summary ...

- Assembler programming without implicit addresses is much too difficult

- The USING instruction tells the assembler which of the sixteen general purpose registers are available for use when it is converting address operands from label to displacement and base

In Summary ...

- One way to decide how to implement the USING instruction is to first determine what object code is desired
- Then labels and USINGs and DROPs can be inserted to generate that object code
- It is easily possible (but maybe not very profitable) to write **all** address operands as implicit

... and Beyond

- Here's a small problem: how can we write the first instruction of our program (STM) with an implicit address rather than 12(13) but without introducing any new labels?

- Here's an answer to analyze on your own:

```
1 APROGRAM CSECT
           USING APROGRAM,13
2 STMINST  STM   14,12,STMINST+12   Save all registers
           DROP  13
           ...
```

- Does the answer work? That is, does it generate the correct object code?

The Answer ...

- ... will be provided to anyone who is interested