Structured Assembler Language Programming Using HLASM

Not Your Father's Assembler Language

Edward E. Jaffe
Phoenix Software International

WAVV Conference
Sunday, May 17, 2009
8 AM Legacy North 1

An Opportunity to Share

I have been a professional Assembler Language programmer for over 26 years.

Along the way, I have made numerous adjustments to my programming methods and style in an effort to become more productive and write better programs.

No adjustment has resulted in a more profound and positive impact than that of adopting a 100% structured programming approach.

I'm honored for the opportunity to share with you.

Structured Programming Disciplines

- Top-down development and design.
  - Program flow is always hierarchical.
  - Levels of abstraction become major routines or separate modules.
  - A module must return to its caller (which could be itself if recursive).
  - Major decision-making appears at as high a level as possible. The routine at the top of the hierarchy is a synopsis of the entire program.
- Programming in which few or no GOTOs are used because only three basic programming structures – mathematically proven to solve *any* logic problem[1] – are used:
  - Sequence.
  - Choice.
  - Repetition.

[1] Corrado Böhm and Guiseppe Jacopini, “Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *Communications of the ACM*, No. 5, May 1966, pp. 366-371.

Other Structured Programming Disciplines Not Discussed

- Team approach.
- Structured walk-throughs.
- Object orientation and organization.
  - Objects.
  - Encapsulation.
  - Inheritance.
  - Classes, Methods, etc.

The Beginning of an Evolution

Prof. Dr. Edsger W. Dijkstra, Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.

*‘For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages …’*

GOTO Density Metric

- The average number of lines of code between two GOTOs.
- Studies show that when sufficiently powerful programming structures are available, GOTOs are not used.
- A 2004 comparison[1] of Fortran programs written in the 1970s to today's C, Ada, and PL8[2] code revealed GOTO densities that differ by several orders of magnitude.
- My research into large assembler language programs showed just under 8 lines per GOTO (branch) not counting subroutine call/return.

	Fortran	**C**	**Ada**	**PL8**	**HLASM**
Files without GOTO	none	81.5%	99.4%	98.5%	none
Lines/GOTO	About 10 [3]	386	13614	1310	<8

[1] W. Gellerich, T. Hendel, R. Land, H. Lehmann, M. Mueller, P. H. Oden, H. Penner, "The GNU 64-bit PL8 compiler: Toward an open standard environment for firmware development", *IBM Journal of Research & Development*, 48, No. 3/4, May/July 2004, pp. 3-4.

[2] PL8 is the language in which much IBM System z firmware is written.

[3] 8% - 13% of all Fortran statements are GOTOs.

Relating GOTO Use to Software Quality

W. Gellerich and E. Plödereder, "The Evolution of GOTO Usage and Its Effects on Software Quality," *Informatik '99*, K. Beiersdörfer, G. Engels, and W. Schäfer, Eds., Springer-Verlag, Berlin, 1999

From Abstract: This paper presents the results of a study in which we analyzed the frequency and typical applications of GOTO in over **400 MB of C and Ada source code**. The frequency analysis showed a large difference in GOTO density. The usage analysis demonstrated that **the availability of sufficiently powerful control structures significantly reduces the frequency of GOTO**. Relating these results to error rates reported for large software projects indicates that **programs written in languages with lower GOTO density are more reliable.**

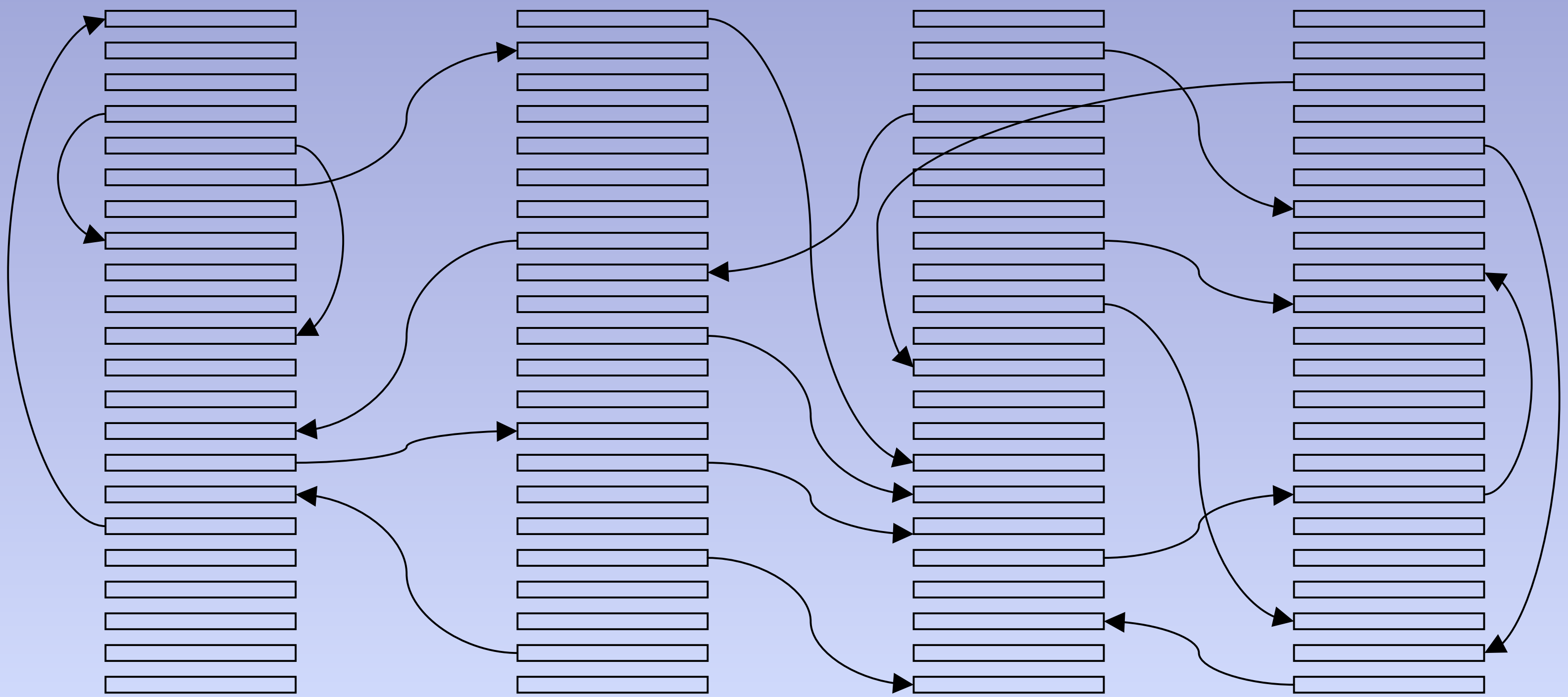
Translation: GOTO statements, when used, remain as harmful today as they were when Dijkstra first warned about them in 1968!

Use of GOTO in Modern Programming Languages – Abolished!

- Most programming languages used today either discourage or completely disallow the use of GOTO statements.
- Those more recently invented are more likely to prohibit its use altogether:
  - Fortran (1957) GOTO is required
  - Basic (1960) GOTO is required
  - C (1973) GOTO is rarely or never used
  - Rexx (1981) GOTO is rarely or never used (not documented)
  - Ada (1983) GOTO is rarely or never used
  - C++ (1985) GOTO is rarely or never used
  - Perl (1987) GOTO is rarely or never used
  - Visual Basic (1991) GOTO is rarely or never used
  - Python (1991) has no GOTO statement
  - Ruby (1993) has no GOTO statement
  - Java (1994) has no GOTO statement

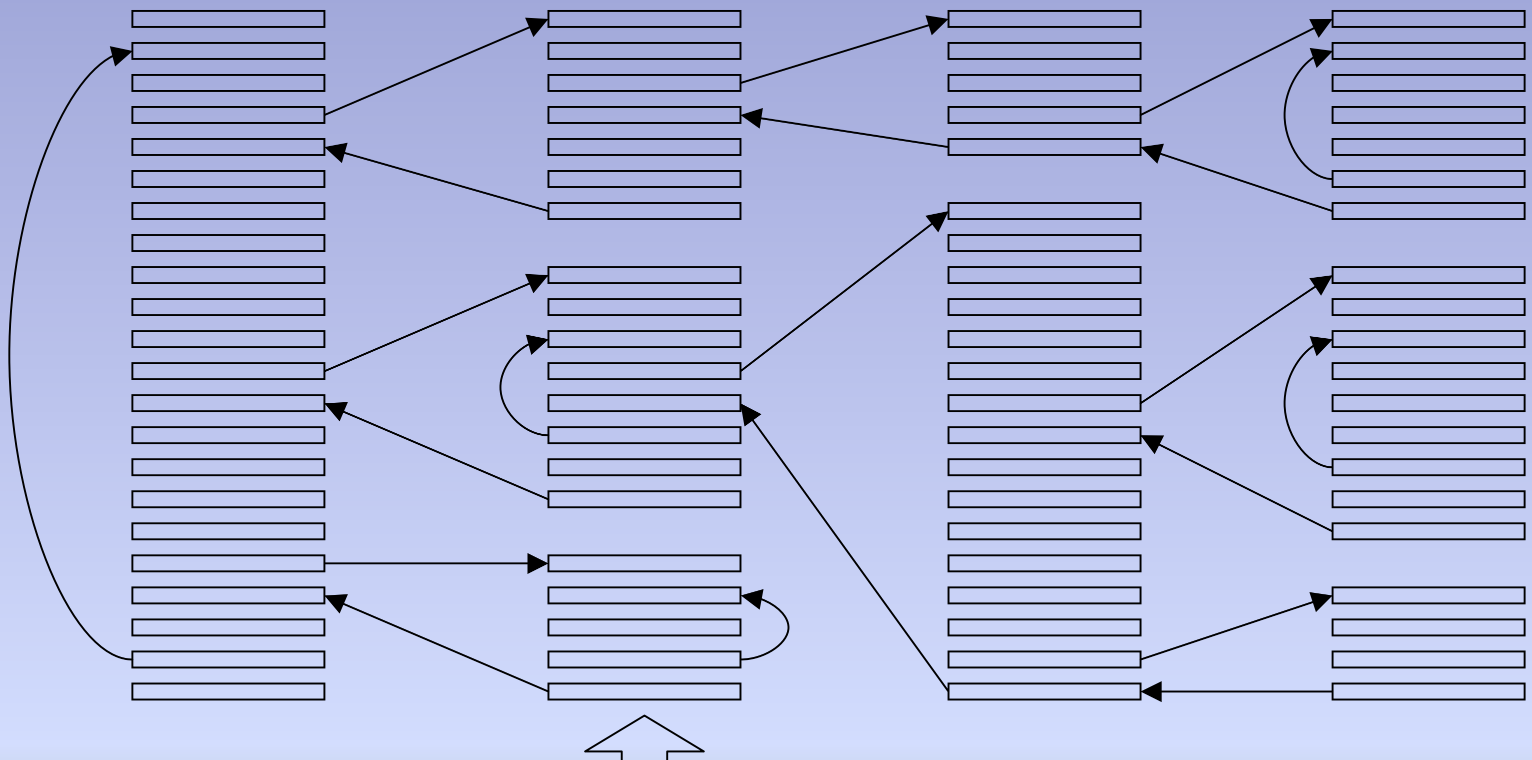
Unstructured Programs: Become Unnecessarily Complex

Customized Program Flow – Can Become “Spaghetti”



Structured Programs: Much Easier to Understand

Hierarchical Program Flow – Building Blocks



Any block in this diagram may be a single statement, construct, subroutine, or an entire subprogram or module.

Structured Programming Using Very Old Languages

- Three articles and a good text book on the subject:
  - Niklaus Wirth, “On the Composition of Well-Structured Programs”, ACM Computing Surveys (CSUR), Vol. 6, No. 4, December 1974, pp. 247-259.
  - Donald E. Knuth, “Structured Programming with go to Statements”, ACM Computing Surveys (CSUR), Vol. 6, No. 4, December 1974, pp. 261-301.
  - Brian W. Kernighan, P. J. Plauger, "Programming Style: Examples and Counterexamples", ACM Computing Surveys (CSUR), Vol. 6, No. 4, December 1974, pp. 303-319.
  - C.E. Hughes, C.P. Pfleeger, and L.L. Rose, “Advanced Programming Techniques. A Second Course in Programming in FORTRAN”, New York, John Wiley and Sons, 1978, ISBN:0-471-02611-5
- The idea is to use GOTO only as a means of implementing control structures. This is necessary in older languages that do not natively implement the control structures.

Structured Programming Entropy in Very Old Languages

- This kind of “structured” programming depends on extra-disciplined programmers making efforts above and beyond the norm.
- Without enforcement from the compiler, the structure of such programs is easily corrupted. Corruption can occur inadvertently by a programmer who doesn’t fully understand the original intent or deliberately by a hurried “quick” fix.
- Human nature being what it is, the path of (apparent) least resistance is almost always taken.
- Thus, superimposed, artificial structure using GOTOs tends to deteriorate over time – a type of increasing entropy – as the program reverts back to its “native,” unstructured state.

Further Stacking the "Deck" Against Mainframe Assembler Language

- Structured programs always contain hierarchical call/return paths. Such a design is best implemented with a low-overhead stacking mechanism for saving/restoring register contents.
- No such mechanism has been provided to assembler language programmers. Even the most simple save area stacking remains a "roll your own" proposition. The hardware linkage stack, introduced with ESA/370, provides only modest relief.
- Without save area stacking, assembler language programs often have a flat, rather than hierarchical, organization.
- This creates much temptation for convoluted logic and/or branches from the middle of one "routine" into another.

Nesting. The Most Important Element of Overall Program Structure

- Nest!
  - Subroutines should not be created just to avoid code duplication. They should be the norm.
  - Subroutines bring <u>order</u> and <u>organization</u>.
- Nest!
  - Implement a low-overhead stacking mechanism.
- Nest!
  - All routines should kept to a manageable size – no more than a couple/few of "pages" of code if possible.
- Don't overdo it!
  - Like everything else in life, there are trade-offs.
  - Gratuitous nesting can affect performance.
  - Choose subroutine boundaries wisely, especially in performance sensitive code.

Well-written, Yet “Flat” Program Organization

```
Task1     DC     0H

* Find the Entry
Find      DC     0H
          L      ...
          L      ...
Find1     DC     0H
          LTR    ...
          JNZ    Find2
          LHI    ...
          J      SetMsg2
Find2     DC     0H
          C      ...
          JE     Process
          L
          J      Find1

* Process the Entry
Process   DC     0H
          MVC    ...
          L      ...
          ST     ...
          TM     ...
          JZ     Process1
          MVI    ...
          J      Account
Process1  DC     0H
          MVC    ...
          MVC    ...
          CLC    ...
          JE     Account
          ST     ...
```

```
* Perform Accounting
Account   DC     0H
          L      ...
          L      ...
          LHI    ...
Account1  DC     0H
          CLI    ...
          JE     SetMsg1
          L      ...
          A      ...
          ST     ...
          TM     ...
          JZ     Account2
          L      ...
          A      ...
          ST     ...
Account2  DC     0H
          LTR    ...
          JH     SetMsg1
          LA     ...
          JCT    xx,Account1
          J      Task2

* Set Messages
SetMsg1   DC     0H
          .
          .
SetMsg2   DC     0H
          .
          .
Task2     DC     0H
```

Hierarchical Program Organization

```
         (mainline)
         .
         JAS    R14,Task1
         JAS    R14,Task2
         .
         .
         STKSAVE POP
         BR     R14

****************************************
*             Perform Task 1           *
****************************************
Task1    DC     0H
         STKSAVE PUSH
         JAS    R14,Task1Find
         LTR    R15,R15
         JNZ    Task1Msg
         JAS    R14,Task1Proc
         LTR    R15,R15
         JNZ    Task1Msg
         JAS    R14,Task1Acct
         J      Task1Ret
Task1Msg DC     0H
         .
         .
Task1Ret DC     0H
         STKSAVE POP
         BR     R14
```

```
Task1Find DC    0H
         STKSAVE PUSH
         .
         .
         STKSAVE POP,
                RETREGS=(R15)
         BR     R14

Task1Proc DC    0H
         STKSAVE PUSH
         .
         .
         STKSAVE POP,
                RETREGS=(R15)
         BR     R14

Task1Acct DC    0H
         STKSAVE PUSH
         .
         .
         STKSAVE POP
         BR     R14

****************************************
*             Perform Task 2           *
****************************************
Task2    DC     0H
         STKSAVE PUSH
         .
         .
         STKSAVE POP
         BR     R14
```

Structured Programming Macros (SPMs)

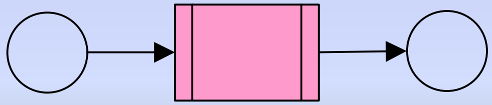
- Leverage powerful HLASM capabilities.
  - HLASM macro support is extremely powerful. Most HLLs – even those that claim to support so-called “macros” – have no equivalent.
- Enforce program structure.
- Eliminate GOTO statements from program source.
- Eliminate extraneous labels.
- Eliminate out-of-line logic paths.
- Enhance source code readability.
- Provide uniformity and standardization – building blocks.
- Provide many HLL benefits without HLL overhead.

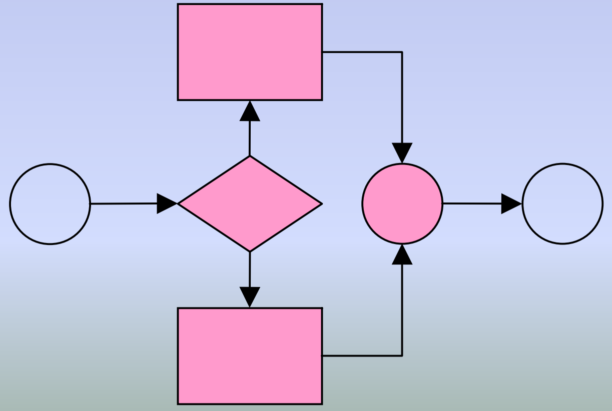
SPMs Enforce Program Structure

- SPMs define the building blocks used to author the program.
- They provide enforcement necessary to prevent corruption of program structure.
- No manually-created, artificial “structure” is imposed on the program source. The program is coded naturally.
- Requires no more programmer cooperation than do HLLs that support GOTO but discourage its use (e.g., Perl or C).

SPMs Eliminate GOTO Statements from Program Source

- As predicted by the studies, SPM use reduces the need/desire to code GOTO (BC and BRC instructions).
- Conditional branching is performed in accordance with the universally-understood rules of the construct. Control *always* returns back to the original path. Branching between constructs is prohibited.
- ***SPMs "hide" the branches that form the constructs.***





SPMs Eliminate Extraneous Labels

- Labels (other than those used for subroutines, labeled USINGs, etc.) represent unstructured exposures. The more labels that exist, the higher the probability that one or more of them will be used as the target of a branch.
- Label management (naming/renaming) is “busy work” and a constant source of programming errors.
- Code fragments copied from one part of a program to another require label “fix up”. Mistakes here can produce loops or worse. BTDTGTTS!
- ***SPMs “hide” the labels that form the constructs.***

SPMs Eliminate Out-of-line Logic Paths

- Out of line logic paths make programs harder to follow.
- *Every* branch is an opportunity to create out-of-line logic.
- Structured programs avoid this pitfall.

```
         LA    R1,Table
         LHI   R0,TableCount
LOOPTOP  DC    0H
         CLI   0(R1),value2
         JE    LABELA
         CLI   0(R1),value3
         JE    LABELB
         CLI   0(R1),value1
         JNE   ITERATE
         .
         . (code for value1)
         .
ITERATE  DC    0H
         LA    R1,TableEntLn(,R1)
         JCT   R0,LOOPTOP
         J     LABELX
LABELA   DC    0H
         .
         . (code for value2)
         .
         J     ITERATE
LABELB   DC    0H
         .
         . (code for value3)
         .
         J     ITERATE
LABELX   DC    0H
```

```
         LA    R1,Table
         LHI   R0,TableCount
LOOPTOP  DC    0H
         CLI   0(R1),value1
         JNE   LABELA
         .
         . (code for value1)
         .
         J     ITERATE
LABELA   DC    0H
         CLI   0(R1),value2
         JNE   LABELB
         .
         . (code for value2)
         .
         J     ITERATE
LABELB   DC    0H
         CLI   0(R1),value3
         JNE   ITERATE
         .
         . (code for value3)
         .
ITERATE  DC    0H
         LA    R1,TableEntLn(,R1)
         JCT   R0,LOOPTOP
LABELX   DC    0H
```

SPMs Enhance Source Code Readability

- SPMs facilitate code indentation – arguably the *single most powerful heuristic ever devised* for illustrating conditional program flow within source code.
- Source code editors on both mainframe and PC are designed to work with indented code such as that typically found in PL/I, C, Pascal, Ada, Visual Basic, REXX, Perl, Ruby, Java, etc.
- Most decent mainframe editor features include:
  - Line commands for shifting columns (to change indentation level).
  - Ability to exclude entire blocks of code from view.
- Some editors (e.g. ISPF) even provide line-oriented editing commands whose behaviors are sensitive to the indentation level of the code.

SPMs Provide Uniformity and Standardization

- SPMs reduce the number of different kinds of constructs used to write the program. They form the building blocks from which the program logic is constructed.
- No “custom” programming constructs are possible.
- Every programmer that reads or modifies the program understands *a priori* the flow of each construct without tedious inspection of the logic.
- Good programmers visualize their programs before they write them. Good programmers that use SPMs will visualize *structured* programs before they write them.
- Programmers learn to solve problems with the tools they are given. Programmers will actually <u>*think*</u> differently!

Which is More Readable/Maintainable?

Try to make an unbiased assessment of the potential for coding mistakes and what's required to add new cases.

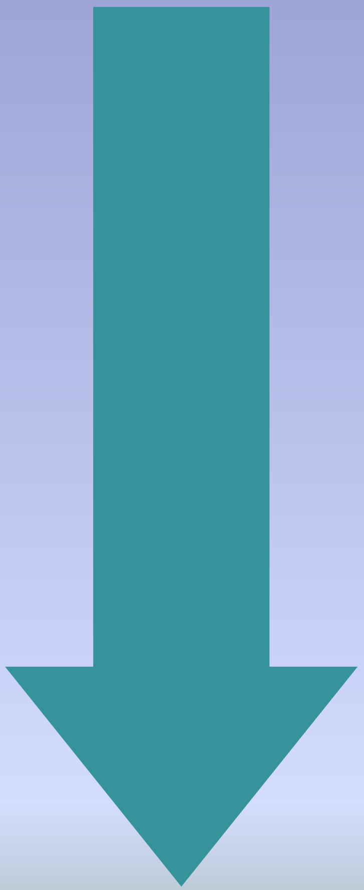
```
         CLI    0(R1),value1
         JNE    LABELA
         .
         . (code for value1)
         .
         J      LABELX
LABELA   DC     0H
         CLI    0(R1),value2
         JNE    LABELB
         .
         . (code for value2)
         .
         J      LABELX
LABELB   DC     0H
         CLI    0(R1),value3
         JNE    LABELC
         .
         . (code for value3)
         .
         J      LABELX
LABELC   DC     0H
         .
         . (handle all other cases)
         .
LABELX   DC     0H
```

```
      SELECT CLI,0(R1),EQ
      WHEN value1
        .
        . (code for value1)
        .
      WHEN value2
        .
        . (code for value2)
        .
      WHEN value3
        .
        . (code for value3)
        .
      OTHRWISE ,
        .
        . (handle all other cases)
        .
      ENDSEL ,
```

Building Blocks

- Single instruction – sequence.
- DO – logic boundary, choice and repetition.
- Programming structures implementing additional “look and feel” to choice and repetition.
- Subroutine.
- Control section.
- Module.

Increasing complexity



Building Blocks – Single Instruction

- The smallest building block.
- Put a few of them together to do something useful.

```
.
.
L      R0,RecCount               Get record count
AHI    R0,1                      Add 1
ST     R0,RecCount               Update record count
.
.
```

Building Blocks – Simple DO

- Define logic start/end boundaries.
  - Imparts order and organization.
- Perform logic tests and controlled branching.
- By far the most useful structure of all.
  - A large, complex program *could* be written using no other structures!

```
>>──DO─────┬──────────────┬──────────────────────────────><
           └─LABEL=label──┘

>>──DOEXIT──condition(s)───┬──────────────┬──────────────><
                           └─DO=dolabel───┘

>>──ASMLEAVE────┬──────────┬─────────────────────────────><
                └─dolabel──┘

>>──ITERATE────┬──────────┬──────────────────────────────><
               └─dolabel──┘

>>──ENDDO────────────────────────────────────────────────><
```

Building Blocks – Simple DO Logic

- This routine updates a record count field when a record exists.
- The ProcessDetail routine is invoked only for records that are not headers or trailers.

```
.
.
DO ,                                 Do for record
  ICM   R3,B'1111',RecPtr              Get record address
  DOEXIT Z                             Exit if no record
  L     R0,RecCount                    Get record count
  AHI   R0,1                           Add 1
  ST    R0,RecCount                    Update record count
  DOEXIT CLI,RecType,EQ,RecHdr         Exit if header
  DOEXIT CLI,RecType,EQ,RecTrl         Exit if trailer
  JAS   R14,ProcessDetail              Process detail record
ENDDO ,                              EndDo for record
.
.
```

- Logic is exactly analogous to what would be traditionally coded. There is no additional overhead whatsoever.

Building Blocks – Simple DO Mainline

- Below is an example of a mainline that calls many subroutines.
- I encapsulate almost every major piece of logic in a simple DO.

```
         .
         .
         DO LABEL=MainLine                Do mainline
           JAS   R14,FindIt                 Locate the instance
           DOEXIT LTR,R15,R15,NZ            Exit if error
           JAS   R14,Modify                 Modify the instance
           DOEXIT LTR,R15,R15,NZ            Exit if error
           JAS   R14,AcctUpdt               Update accounting info
           DOEXIT LTR,R15,R15,NZ            Exit if error
           JAS   R14,Unlock                 Unlock the data base
           DOEXIT LTR,R15,R15,NZ            Exit if error
           JAS   R14,Report                 Generate report data
           DOEXIT LTR,R15,R15,NZ            Exit if error
           .
           . (Insert additional calls here)
           .
         ENDDO , MainLine                 EndDo mainline
         .
         .
```

- Again, exactly analogous to traditional code. But, without the ever-present temptation to branch outside the structure.

Building Blocks – Simple DO Looping

- This simple DO drives a loop to repetitively process “entries”.
- ITERATE is used to perform the looping.

```
.
.
DO ,                             Do for all entries
  JAS   R14,GetEntry               Get the next entry
  DOEXIT LTR,R15,R15,NZ            Exit if no more entries
  JAS   R14,ProcessEntry           Process the entry
  ITERATE ,                        Process next entry
ENDDO ,                          EndDo for all entries
.
.
```

Building Blocks – Nested Simple DO

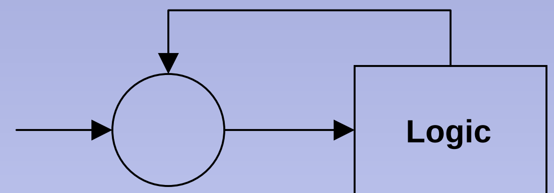
- Implement more complex choice logic.

```
.
.
DO LABEL=SetVarsMsg           Do for msg processing

  DO ,                          Do for msg include tests
    DOEXIT CLI,CurMsgType,LE,C' ' Include if no msg yet formatted
    DOEXIT TM,MsgFlgs,Error,O     Include if an error message
    .                             .
    . (other include tests)       .
    .                             .
    ASMLEAVE SetVarsMsg           Bypass message formatting
  ENDDO ,                       EndDo for msg include tests

  .
  . (format the message to be displayed)
  .

ENDDO , SetVarsMsg            EndDo for msg processing
.
.
```

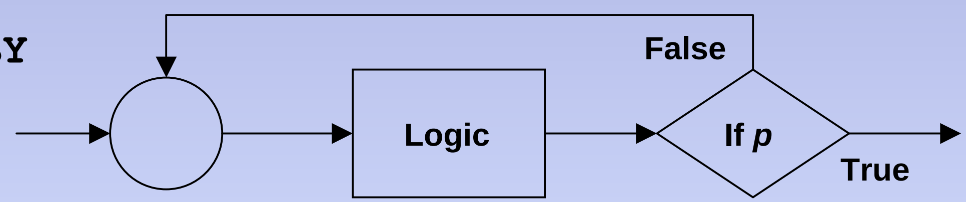
Building Blocks – More DO Keywords

- Additional DO keywords provide more looping choices.
- BCT/JCT, BXH/JXH, BXLE/JXLE loops are supported.

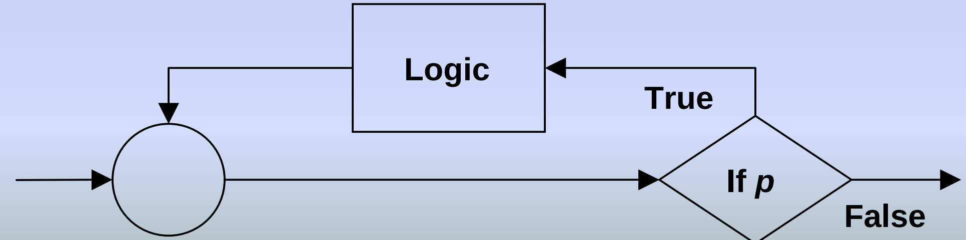
DO INF



DO UNTIL or FROM, TO, BY



DO WHILE



Building Blocks – SELECT

- Test conditions sequentially.
- When condition is true, perform appropriate logic and then exit the structure.
  - NEXTWHEN statement may be used within WHEN clause to continue testing remaining conditions rather than exiting the structure.
- Optional “otherwise” clause.

```
>>──SELECT──┬──────────────────┬──────────────────────────────────><
            └─condition-prefix─┘

>>──WHEN────┬─value────────┬──────────────────────────────────────><
            └─condition(s)─┘

>>──NEXTWHEN──────────────────────────────────────────────────────><

>>──OTHRWISE──────────────────────────────────────────────────────><

>>──ENDSEL────────────────────────────────────────────────────────><
```

Building Blocks – SELECT

- This code fragment takes various actions based on the contents of register 1 (the so-called "start" value).
- In case you're wondering about the ASMLEAVE, this real-world SELECT was nested inside a simple DO (of course)!

```
.
.
SELECT ,                              Select Start value
WHEN CHI,R1,EQ,0                      When Start=FIRST
  MVC   EMRPARMS,=F'1'                  Force to top of data
WHEN CL,R1,EQ,=X'7FFFFFFF'            When Start=LAST (explicit)
  MVC   EMRPARMS,=X'7FFFFFFF'           Set both values to LAST
  MVC   EMRPARMS+4,=X'7FFFFFFF'       (same)
  ASMLEAVE ,                            Exit the structure
WHEN CHI,R1,EQ,-1                     When Start=Current
  MVC   EMRPARMS,CBLKATNM               Set to absolute number at top
WHEN CHI,R1,EQ,-2                     When Start=Time/Date (unsupported)
  MVC   EMRPARMS,=F'1'                  Force to top of data
WHEN CHI,R1,LT,0                      When Start=Label
  MVC   EMRPARMS,0(R1)                  Set value at label
OTHRWISE ,                            Otherwise Start=ordinary numeric
  AL    R1,CBLKBNDL                     Make relative to low boundary
  AHI   R1,-1                            (same)
  ST    R1,EMRPARMS                      (same)
ENDSEL ,                              EndSel Start value
.
.
```

Building Blocks – CASE

- An implementation of the familiar “branch table” – used to associate program logic with uniformly-distributed numeric values.
- Handles values >0 and some power of 2.

```
                    ┌─POWER=0─────┐
>>──CASENTRY──┼─────────────┼──────────────────────────────────><
                    └─POWER=power─┘

           <─,───┐
>>──CASE──value──┴─────────────────────────────────────────────><

>>──ENDCASE────────────────────────────────────────────────────><
```

Building Blocks – CASE

- This code fragment invokes a different routine depending on the value of the “service call code” loaded into R14.

```
.
.
CASENTRY R14                          Cases for service call
CASE 1                                EAPDCallGetInput
  JAS   R14,APIS_GetInput               Get caller input
  STM   R0,R1,TSAREG00                  Pass back length & ptr
CASE 2                                EAPDCallSetMsg
  JAS   R14,APIS_SetMsg                 Set messages
CASE 3                                EAPDCallSetScrn
  JAS   R14,APIS_SetScrn                Set screen data
CASE 4                                EAPDCallSetFunc
  JAS   R14,APIS_SetFunc                Set function data
CASE 5                                EAPDCallSetPos
  JAS   R14,APIS_SetPos                 Set position data
CASE 6                                EAPDCallTermNtfy
  JAS   R14,APIS_TermNtfy               Notify API of termination
ENDCASE ,                             EndCases for service call
.
.
```

Building Blocks – IF

- Implementation of familiar IF/THEN/ELSE choice structure.
- ELSE and ELSEIF are optional.
- ELSEIF may be used to create a structure similar to SELECT.
- Numerous logical connectors available for compound tests.

```
>>──IF──condition(s)───┬──────┬──────────────────────────────────────><
                       └─THEN─┘
>>──ELSEIF──condition(s)───┬──────┬──────────────────────────────────><
                           └─THEN─┘
>>──ELSE─────────────────────────────────────────────────────────────><
>>──ENDIF────────────────────────────────────────────────────────────><
```

Building Blocks – IF

- This code fragment obtains the “job” name in a z/OS environment from pointers in an ASCB control block.

```
.
.
L     R14,PSAAOLD                    Load ASCB address
USING ASCB,R14                   *** Synchronize ASCB
LT    R15,ASCBJBNI                   Load address of job name
IF NZ                                If job name available
  MVC   ESMFJOBN,0(R15)                Set job name
ELSE ,                               Else
  LT    R15,ASCBJBNS                   Load address of task name
  IF NZ                                If task name available
    MVC   ESMFJOBN,0(R15)                Set as job name
  ELSE ,                               Else
    MVC   ESMFJOBN,=C'*UNKNOWN'          Set name to '*UNKNOWN'
  ENDIF ,                              EndIf
ENDIF ,                              EndIf job name available
DROP  R14                        *** Drop ASCB
.
.
```

Building Blocks – Subroutine

- Subroutines bring <u>order</u> and <u>organization</u>.
- Logic boundaries are created.
- Source code indentation starts over.
- A “legitimate” use for a label.
- R14 is normally used to hold the return address.
- Generally, a return code (if any) is passed back in R15. There may also be pointers, counts, tokens, etc. passed back in R1 and R0.
- Very local subroutines often use and/or pass back additional registers.

Building Blocks – Subroutine

```
*****************************************************************
* Invoke IRXEXCOM to Update Variables                           *
*****************************************************************
REXX_SetVarsEXCOM DC 0H
          CL     R4,Dws_VarArea              Update requested ?
          BNHR   R14                         Return if not
          EJESSRV TYPE=STKPUSH,              Save the registers
                 REGS=(R14:R1)               - Regs to save/restore
          MVCIN  Dws_WorkD1,=C'MOCXEXRI'+7 Set IRXEXCOM char value
          LA     R0,Dws_WorkD1               Set parameter #1
          ST     R0,Dws_MacWk+00             (same)
          LA     R0,=F'0'                    Set parameters #2 & #3
          ST     R0,Dws_MacWk+04             (same)
          ST     R0,Dws_MacWk+08             (same)
          MVC2   Dws_MacWk+12,Dws_VarArea Set parameter #4
          OI     Dws_MacWk+12,X'80'          Indicate end of list
          L      R15,Dws_EnvBlock            Get EnvBlock address
          LR     R0,R15                      Pass EnvBlock ptr in R0
          L      R15,ENVBLOCK_IRXEXTE-ENVBLOCK(,R15) Get IRXEXTE address
          LA     R1,Dws_MacWk                Point to parm list
          L      R15,IRXEXCOM-IRXEXTE(,R15) Get IRXEXCOM address
          BASR   R14,R15                     Invoke IRXEXCOM service
          LM     R4,R5,Dws_VarArea           Get variable area ptr/length
          XC     Dws_VarPtr,Dws_VarPtr       Zero last variable pointer
          EJESSRV TYPE=STKPOP                Restore the registers
          BR     R14                         Return
          DROP   R7                      *** Drop Var_
```

Combining SPM Condition Tests With Instructions That Set the CC

```
.
.
L     R14,GENASCB                     Load ASCB address
USING ASCB,R14                    *** Synchronize ASCB
IF LT,R15,ASCBJBNI,NZ                 If job name available
  MVC   ESMFJOBN,0(R15)                 Set job name
ELSE ,                                Else
  IF LT,R15,ASCBJBNS,NZ                 If task name available
    MVC   ESMFJOBN,0(R15)                 Set as job name
  ELSE ,                                Else
    MVC   ESMFJOBN,=C'*UNKNOWN'           Set name to '*UNKNOWN'
  ENDIF ,                               EndIf
ENDIF ,                               EndIf job name available
DROP  R14                         *** Drop ASCB
.
.
```

Combining SPM Condition Tests With Macros That Set the CC

```
MACRO ,
$NSXENCL ,
$NSXCALL PCVTSSEOT,PARMS=SET  Invoke enclave eligibility
LTR   R15,R15                 Test return code
MEND  ,
.
.
.
IF $NSXENCL,0,0,NZ
  JAS   R14,BadEnclaveSet
ELSE ,
  .
  . (process logic in enclave)
  .
ENDIF ,
```

*Thanks to Tom Harper for pointing this out!

Enabling Use of the SPMs

- Update SYSLIB concatenation:
  - HLA.SASMMAC2 for z/OS
  - PRD2.PROD for z/VSE
- Add the following to the top of your program:

```
COPY  ASMMSP                         Structured Assembler Support
```

- Add the following if your program uses relative branch instructions:

```
ASMMREL ON                           Enable relative branch for SPMs
```

- z/OS users should add one of the following as well:

```
SYSSTATE ARCHLVL=1                   Program supports immediate/relative
      -OR-
SYSSTATE ARCHLVL=2                   Program supports z/Architecture
```

Customizing the Macro Names

Make Modifications to IBM macro ASMMNAME

```
&ASMA_NAMES_CASE     SETC 'CASE'                                        00044000
&ASMA_NAMES_CASENTRY SETC 'CASENTRY'                                    00045000
&ASMA_NAMES_DO       SETC 'DO'                                          00046000
&ASMA_NAMES_DOEXIT   SETC 'DOEXIT'                                      00047000
&ASMA_NAMES_ELSE     SETC 'ELSE'                                        00048000
&ASMA_NAMES_ENDCASE  SETC 'ENDCASE'                                     00049000
&ASMA_NAMES_ENDDO    SETC 'ENDDO'                                       00050000
&ASMA_NAMES_ENDIF    SETC 'ENDIF'                                       00051000
&ASMA_NAMES_ENDLOOP  SETC 'ENDLOOP'                                     00052000
&ASMA_NAMES_ENDSEL   SETC 'ENDSEL'                                      00053000
&ASMA_NAMES_ENDSRCH  SETC 'ENDSRCH'                                     00054000
&ASMA_NAMES_EXITIF   SETC 'EXITIF'                                      00055000
&ASMA_NAMES_IF       SETC 'IF'                                          00056000
&ASMA_NAMES_ORELSE   SETC 'ORELSE'                                      00057000
&ASMA_NAMES_OTHRWISE SETC 'OTHRWISE'                                    00058000
&ASMA_NAMES_SELECT   SETC 'SELECT'                                      00059000
&ASMA_NAMES_STRTSRCH SETC 'STRTSRCH'                                    00060000
&ASMA_NAMES_WHEN     SETC 'WHEN'                                        00061000
&ASMA_NAMES_ELSEIF   SETC 'ELSEIF'                                      00062000
&ASMA_NAMES_LEAVE    SETC 'LEAVE'                                   EEJ 00063000
&ASMA_NAMES_ITERATE  SETC 'ITERATE'                                     00064000
&ASMA_NAMES_NEXTWHEN SETC 'NEXTWHEN'                                    00065000
```

Getting SPMs Inside Macros to Print

- The SPMs explicitly *disable* printing of their own inner macro calls using PRINT NOMCALL.
- Enable printing of inner macro calls using PRINT MCALL to ensure SPM invocations appear on the assembler listing.

```
          MACRO ,
          TESTMAC ,
      PUSH PRINT,NOPRINT                 <Save PRINT status>
      PRINT MCALL,NOPRINT                <Print macro calls>
      XR    R15,R15                      Set return code = 0
      IF CLI,0(R1),EQ,C'X'               If R1 points to 'X'
        LHI   R15,4                        Set return code = 4
      ENDIF ,                            EndIf
      POP   PRINT,NOPRINT                <Restore PRINT status>
          MEXIT ,
          MEND  ,
```

```
      TESTMAC ,
+     XR    R15,R15                      Set return code = 0
+     IF CLI,0(R1),EQ,C'X'               If R1 points to 'X'
+         CLI         0(R1),C'X'
+         BRC         15-8,#@LB1
+       LHI   R15,4                        Set return code = 4
+     ENDIF ,                            EndIf
+#@LB1    DC    0H
```

The Source Record Layout I Use

- Long (but reasonable) labels used for major routines.
- Short labels (4 chars or less) for labeled USINGs.
- “Zero-indent” operation code begins in column 6.
- “Zero-indent” operand begins in column 12.
- “Zero-indent” commentary begins in column 36.
- Indentation delta is always 2 bytes.
- Comment blocks for subroutines start in column 1.
- Small comment blocks for code fragments follow indentation.

The Source Record Layout I Use

```
         1         2         3         4         5         6         7
123456789012345678901234567890123456789012345678901234567890123456789012
****************************************************************
*                                                              *
*                   Perform UNIT Modifications                 *
*                                                              *
****************************************************************
ModifyUnit DC 0H
     STKSAVE PUSH                 Save the registers

     BASR  R12,0                  Point to constants
     AHI   R12,ModifyUnitConst-*  (same)
     USING ModifyUnitConst,R12 *** Synchronize base register

   ************************************
   * Get Specified Value              *
   ************************************
     MVI   LIFLDTID,EFLTLIUN      Set field text unit ID
     EJESSRV TYPE=GETBOVR,        Get batch overtype value
           PARM=EFLTLIUN          (same)

     XR    R15,R15                Zero out message number

     IF CLI,LIUNIT,GT,C' '        If value supplied

     ************************************
     * Validate the Value               *
     ************************************
       DO ,                         Do for validation
         IF CLI,LIUNIT,EQ,C'S'        If SNA requested
           MVC2  LIUNIT,=CL4'SNA'       Set to SNA
           ASMLEAVE ,                   Done with validation
         ENDIF ,                      EndIf SNA requested
         .
         . (more code follows …)
         .
```

Some of My Rules of Thumb

- Avoid the use of vectored returns.
  - Vectored returns imply a branch table follows the subroutine linkage.
  - Branch tables imply GOTOs (branches) and labels.
- Try to make USING/DROP and PUSH/POP happen at the same indentation level.
- Use VECTOR=B for CASE macro set when using based branches. (Or just always use relative branches.)
- Choose constructs that require minimal changes to add new cases in the future.
  - Think about the next programmer – even if it's you!
- Avoid excessive indentation.

Some of My Rules of Thumb

- Don’t be afraid to insert “white space” between statements.
- Use large screens when editing (I now use 90x80).
  - The larger the screen, the more logic you can see at once.
  - The more code you can see, the better you understand the “flow”.
  - A “page” of code is whatever size *you* decide it should be. Not just what fits on a sheet of paper. (Does anyone print listings anymore?)
- Keep the size of constructs “reasonable”.
  - Ideally, a construct will fit on one “page” so you can see the boundaries. A couple/few “pages” is not unreasonable.
  - Very large CASE or SELECT structures should have a comment block precede each CASE/WHEN clause. That clause can be about the size of any other “normal” routine.
  - Create subroutines when things start to get unwieldy.

Avoidance of Excessive Indentation

- Rather than nesting many, many IF/THEN constructs (essentially ANDing the outcome of multiple tests):
  - Use simple DO with DOEXIT/ASMLEAVE.
- Rather than nesting many, many IF/ELSE constructs:
  - Use ELSEIF.
  - Use SELECT.
  - Use simple DO with DOEXIT/ASMLEAVE.
- Use subroutines even for code used only once:
  - All subroutines begin at “zero” indentation level.
  - Calling routines become smaller; more readable and maintainable.
  - But don’t overdo it! Save/restore overhead should be minimal compared to the work you are doing in the subroutine.

Challenges Caused by Assembler Language Syntax Restrictions

- Existing assembler language syntax rules are not conducive to free-form indentation.
  - Continuation characters must appear in column 72.
  - Continued statements must begin in column 16.
  - Comment statements must have an asterisk (*) in column 1.
- Shifting a block of code left or right to change the indentation level often creates syntax errors.
- My FLOWASM HLASM exit helps address these issues.

Assembler Language Programming Resources I've Made Public

- Modifications to the SPMs:
  - NEXTWHEN macro (not needed for HLASM 1.6).
  - Carry and borrow condition checking.
- STKSAVE Macro.
  - A macro for managing a save area stack.
  - Based on – but not actually the same as – a macro we use internally.
- FLOWASM HLASM Exit.
  - Allows assembler language programs to be coded naturally with a more free-form syntax.
  - Prints "flow bars" to match up SPMs on the listing.
  - This is *exactly* the same exit we, and some other ISVs, use internally.

Available from:
ftp://ftp.phoenixsoftware.com/pub/demo/flowasm.xmi
ftp://ftp.phoenixsoftware.com/pub/demo/flowasm.zip

STKSAVE Macro

- Low-overhead *local* save area stack services.
- Can optionally save/restore access registers.
- Can save/restore any subset of registers.
- Requires 32-byte stack control area.
  - Initialized by INIT call at program startup.
- Currently for 24/31-bit mode only.

FLOWASM HLASM Exit

- Works on z/OS, z/VM and z/VSE.
- Relaxes cumbersome syntax rules:
  - Comment blocks may start in any column. They may begin with either an asterisk (*) or a slash and asterisk (/*).
  - No explicit continuation needed when macro operand ends with trailing comma.
  - Continued macro operands may start in any column.
- For z/OS, supports both fixed and variable length source input:
  - Variable length input may be numbered or unnumbered.
  - Variable length explicit continuation is trailing ‘+’ character.
  - Library (SYSLIB) input still restricted to LRECL=80.
  - We use only RECFM=FB LRECL=80 source libraries.
- Prints “flow” bars to match up SPMs on the listing.

FLOWASM HLASM Exit

- Reformatting too-long lines:
  - Remove superfluous blanks between op-code and operand.
  - If still too long, remove superfluous blanks between operand and commentary.
  - If still too long, remove superfluous blanks before op-code.
  - If still too long:
    - If operand fits on the line, commentary is truncated.
    - If operand is too long, it is wrapped and continued in column 16 of the next line along with the commentary.
- Automatic continuation:
  - Detects trailing comma on macro operand and supplies ‘-’ continuation character.
  - Continued operand shifted into column 16.
    - If commentary must be moved, it is moved immediately after operand.
    - If line too long, reformat as described above.

HLASM Listing With “Flow” Bars

```
.                                             58489 ****************************************************************
.                                             58490 * Search for Matching Column Name                                *
.                                             58491 ****************************************************************
.0000325C 9200 83FC      000003FC             58492      MVI   SUBSWKH3,X'00'            Zero field TID value
.                                             58493      DO ,                            Do for column name search
.00003260 48E0 83F8               000003F8    58503      | LH    R14,SUBSWKH1              Get normalized length
.00003264 12EE                                58504      | DOEXIT LTR,R14,R14,NP           Exit if invalid length
.0000326A A7EE 0008               00000008    58517      | DOEXIT CHI,R14,GT,L'SUBSWKD1    Exit if too long
.00003272 D207 81C8 C4E8 000001C8 00003530    58530      | MVC   SUBSWKD1,=CL8' '          Blank out work field
.00003278 A7EA FFFF               FFFFFFFF    58531      | AHI   R14,-1                    Make relative to zero
.0000327C 44E0 C4DA               00003522    58532      | EX    R14,MCLCOMV2              Copy to SUBSWKD1
.00003280 43E0 6000               00000000    58533      | IC    R14,EFLLSTID              Get list identifier
.00003284 A7EE 00C0               000000C0    58534      | IF CHI,R14,LT,EFLLSTIB          If tabular utility
.0000328C 06E0                                58548      | : BCTR  R14,0                     Make relative to zero
.0000328E 5810 C4F0               00003538    58549      | : L     R1,=A(JJTUFLDIDX)         Point to index table
.00003292 A7F4 000E               000032AE    58550      | ELSE ,                          Else
.00003296 A7EA FF40               FFFFFF40    58558      | : AHI   R14,-EFLLSTIB             Make relative to base
.0000329A 95F2 A00B      0000000B             58559      | : IF CLI,EMRJES,EQ,EMRJES2        If running JES2
.000032A2 5810 C4F4               0000353C    58573      | : | L     R1,=A(J2TDFLDIDX)         Point to index table
.000032A6 A7F4 0004               000032AE    58574      | : ELSE ,                          Else running JES3
.000032AA 5810 C4F8               00003540    58582      | : | L     R1,=A(J3TDFLDIDX)         Point to index table
.                                             58583      | : ENDIF ,                         EndIf
.                                             58590      | ENDIF ,                         EndIf tabular utility
.000032AE 89E0 0003               00000003    58597      | SLL   R14,3                     Point to proper entry
.000032B2 1EE1                                58598      | ALR   R14,R1                    (same)
.000032B4 98EF E000               00000000    58599      | LM    R14,R15,0(R14)            Get offset & entry count
.000032B8 1EE1                                58600      | ALR   R14,R1                    Change offset into pointer
.                                             58601      | DO FROM=(R15)                   Do for all entries
.000032BA D507 81C8 E000 000001C8 00000000    58614      | : DOEXIT CLC,SUBSWKD1,EQ,0(R14) Exit if matching entry
.000032C4 A7EA 0009               00000009    58627      | : AHI   R14,FLD_TblLen            Advance pointer
.000032C8 A7F6 FFF9               000032BA    58628      | ENDDO ,                         EndDo for all entries
.000032CC 12FF                                58638      | DOEXIT LTR,R15,R15,Z            Exit if column not found
.000032D2 D200 83FC E008 000003FC 00000008    58651      | MVC   SUBSWKH3(1),8(R14)        Copy field TID value
.                                             58652      ENDDO ,                         EndDo for column name search
```

Everything beyond this point is for reference only. It is not part of the material to be presented.

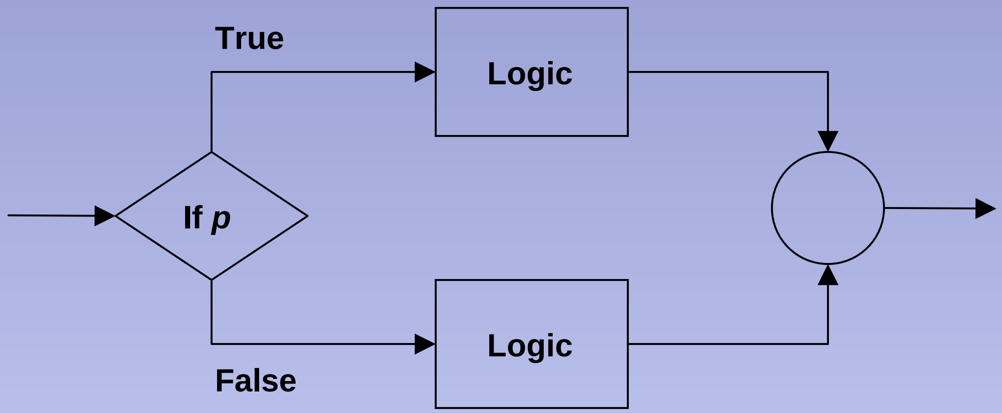
Structured Programming Macro Sets

- IF
- DO
- CASE
- SELECT
- SEARCH

Disclaimer:

There are some coding fragments shown in this presentation. Rather than searching for real-world examples, I made many of them up "on the fly" to illustrate the usage of a particular construct. Consequently, some of the fragments do not make sense. Sorry.

IF Macro Set



Predicate Values	Connectors
Numeric value (1-14) Condition mnemonic Instruction,p1,p2,condition Compare-instruction,p1,condition,p2	AND OR ANDIF ORIF ELSEIF

IF – Mnemonics and Complements

Case	Condition Mnemonics	Meaning	Complement
After compare instructions	H, GT L, LT E, EQ	High, Greater than Low, Less than Equal	NH, LE NL, GE NE
After arithmetic instructions	P M Z O	Plus Minus Zero Overflow	NP NM NZ NO
After test under mask instructions	O M Z	Ones Mixed Zeros	NO NM NZ

IF – Basic Tests

```
IF CLI,0(R1),GT,C' '
  ST    R1,NBPtr
ENDIF ,
```

```
+        CLI    0(R1),C' '
+        BRC    15-2,#@LB1
         ST     R1,NBPtr
+#@LB1   DC     0H
```

```
IF CLI,0(R2),GT,C' '
  ST    R2,NBPtr
ELSE ,
  ST    R2,BPtr
ENDIF ,
```

```
+        CLI    0(R2),C' '
+        BRC    15-2,#@LB3
         ST     R2,NBPtr
+        BRC    15,#@LB5
+#@LB3   DC     0H
         ST     R2,BPtr
+#@LB5   DC     0H
```

IF – Combined Tests

```
IF CLI,0(R1),GE,C'0',AND,
   CLI,0(R1),LE,C'9'
  OI     Flag,Numeric
ENDIF ,
```

```
+        CLI   0(R1),C'0'
+        BRC   15-11,#@LB6
+        CLI   0(R1),C'9'
+        BRC   15-13,#@LB6
         OI    Flag,Numeric
+#@LB6   DC    0H
```

```
IF CLI,0(R1),LT,C'0',OR,
   CLI,0(R1),GT,C'9'
  NI     Flag,X’FF’-Numeric
ENDIF ,
```

```
+        CLI   0(R1),C'0'
+        BRC   4,#@LB9
+        CLI   0(R1),C'9'
+        BRC   15-2,#@LB8
+#@LB9   DC    0H
         NI    Flag,X'FF'-Numeric
+#@LB8   DC    0H
```

IF – Logical Grouping With ANDIF

Note use of optional surrounding parentheses.

```
IF (CLI,0(R1),GT,C' '),OR,
   (LTR,R4,R4,NZ),AND,
   (CLC,SpecChar(2),EQ,0(R4)),
  ANDIF,
   (TM,Flag,FlagBit,NZ),AND,
   (CLM,R15,B'0011',LT,Limit),OR,
   (ICM,R2,B'1111',Offset,Z)
  OI    Flag,Passed
ENDIF ,
```

```
+        CLI    0(R1),C' '
+        BRC    2,#@LB11
+        LTR    R4,R4
+        BRC    15-7,#@LB10
+        CLC    SpecChar(2),0(R4)
+        BRC    15-8,#@LB10
+#@LB11  DC     0H
+        TM     Flag,FlagBit
+        BRC    15-7,#@LB10
+        CLM    R15,B'0011',Limit
+        BRC    4,#@LB12
+        ICM    R2,B'1111',Offset
+        BRC    15-8,#@LB10
+#@LB12  DC     0H
         OI     Flag,Passed
+#@LB10  DC     0H
```

IF – Logical Grouping With ORIF

```
IF (CLI,0(R1),GT,C' '),OR,
   (LTR,R4,R4,NZ),AND,
   (CLC,SpecChar(2),EQ,0(R4)),
  ORIF,
   (TM,Flag,FlagBit,NZ),AND,
   (CLM,R15,B'0011',LT,Limit),OR,
   (ICM,R2,B'1111',Offset,Z)
  OI    Flag,Passed
ENDIF ,
```

```
+        CLI   0(R1),C' '
+        BRC   2,#@LB14
+        LTR   R4,R4
+        BRC   15-7,#@LB13
+        CLC   SpecChar(2),0(R4)
+        BRC   8,#@LB14
+#@LB13 DC    0H
+        TM    Flag,FlagBit
+        BRC   15-7,#@LB15
+        CLM   R15,B'0011',Limit
+        BRC   4,#@LB14
+        ICM   R2,B'1111',Offset
+        BRC   15-8,#@LB15
+#@LB14 DC    0H
         OI    Flag,Passed
+#@LB15 DC    0H
```

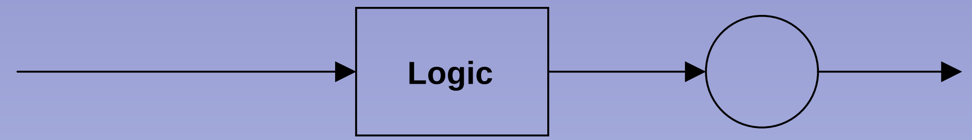
IF – Nesting With ELSEIF

```
IF (CLI,0(R1),EQ,C'0')
  LA    R15,12
ELSE ,
  IF (CR,R2,EQ,R3)
    LA    R15,16
  ELSE ,
    IF CLC,=Y(Big),GT,Size
      LA    R15,24
    ELSE ,
      XR    R15,R15
    ENDIF ,
  ENDIF ,
ENDIF ,
```

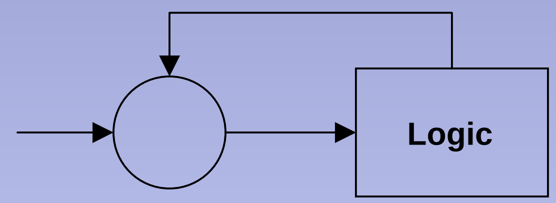
```
IF (CLI,0(R1),EQ,C'0')
  LA    R15,12
ELSEIF (CR,R2,EQ,R3)
  LA    R15,16
ELSEIF CLC,=Y(Big),GT,Size
  LA    R15,24
ELSE ,
  XR    R15,R15
ENDIF ,
```

DO Macro Set

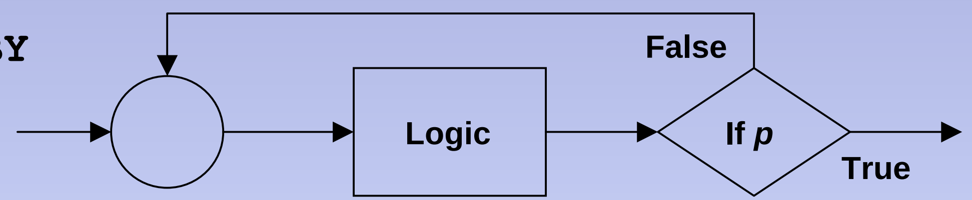
DO ,



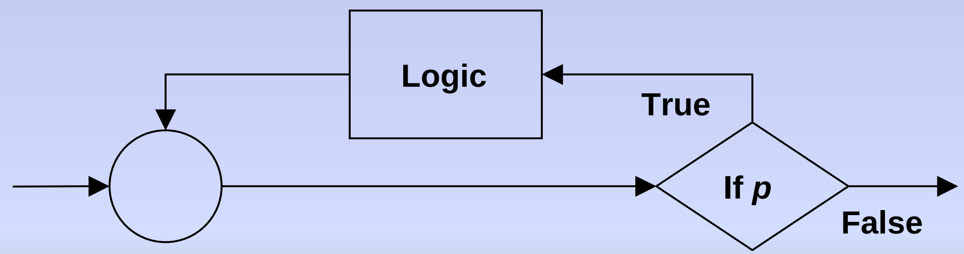
DO INF



DO UNTIL or FROM, TO, BY



DO WHILE



DO – Loop Terminator Generation

Type	Keywords	Other Conditions	Result
Simple	None	ONCE parameter or no parameters (null comma)	No terminator
Infinite loop	Neither FROM, WHILE, nor UNTIL	INF parameter	BC 15 BRC 15
Explicit Specification	FROM, plus TO and/or BY	BXH/BRXH parameter BXLE/BRXLE parameter	BXH, BRXH BXLE, BRXLE
Counting	FROM only	Two or three values	BCT, BCTR BRCT, BRCTR
Backward Indexing	FROM, TO and BY	FROM and TO numeric, FROM value > TO value	BXH BRXH
Backward Indexing	FROM BY	BY numeric and less than zero	BXH BRXH
Forward Indexing	All other combinations		BXLE BRXLE

DO – Register Initialization

Value Given	Instruction Generated
None	None (passed in)
Zero	SR Rx,Rx
0 to 4095	LA Rx,value
-32768 to –1 or 4096 to 32767	LHI Rx,value or LH Rx,=H’value’
Other numbers	L Rx,=F’value’
(value)	LR Rx,value
Other	L Rx,Other

DO – Basic Formats

Simple

```
DO ,
  JAS   R14,ProcessInput
ENDDO ,
```

```
+#@LB21 DC     0H
        JAS    R14,ProcessInput
```

Infinite

```
DO INF
  JAS   R14,ProcessTillDead
ENDDO ,
```

```
+#@LB18 DC     0H
        JAS    R14,ProcessTillDead
+       BRC    15,#@LB18
```

DO – Backward Index (Implied BXH)

```
DO FROM=(R1,100),TO=(R5,1),
   BY=(R4,-1)
  STC   R1,0(R1,R2)
ENDDO ,
```

```
+       LA    R1,100
+       LA    R5,1
+       LHI   R4,-1
+#@LB38 DC    0H
        STC   R1,0(R1,R2)
+#@LB39 DC    0H
+       BRXH  R1,R4,#@LB38
```

```
DO FROM=(R1,100),BY=(R5,-1)
  STC   R1,0(R1,R2)
ENDDO ,
```

```
+       LA    R1,100
+       LHI   R5,-1
+#@LB41 DC    0H
        STC   R1,0(R1,R2)
+#@LB42 DC    0H
+       BRXH  R1,R5,#@LB41
```

DO – Forward Index (Implied BXLE)

```
DO FROM=(R1,1),TO=(R5,100),
   BY=(R4,1)
  STC   R1,0(R1,R2)
ENDDO ,
```

```
+        LA    R1,1
+        LA    R5,100
+        LA    R4,1
+#@LB47  DC    0H
         STC   R1,0(R1,R2)
+#@LB48  DC    0H
+        BRXLE R1,R4,#@LB47
```

```
DO FROM=(R1,ArrayFirst),
   TO=(R5,ArrayLast),
   BY=(R4,=A(EntryLen))
  JAS   R14,ProcessEntry
ENDDO ,
```

```
+        L     R1,ArrayFirst
+        L     R5,ArrayLast
+        L     R4,=A(EntryLen)
+#@LB44  DC    0H
         JAS   R14,ProcessEntry
+#@LB45  DC    0H
+        BRXLE R1,R4,#@LB44
```

DO – Explicit BXH/BXLE

I recommend the use of explicit BXH/BXLE specification

```
DO BXLE,FROM=(R1,1),TO=(R15,100),
        BY=(R14,1)
  STC   R1,0(R1,R2)
ENDDO ,
```

```
+          LA     R1,1
+          LA     R15,100
+          LA     R14,1
+#@LB32    DC     0H
           STC    R1,0(R1,R2)
+#@LB33    DC     0H
+          BRXLE  R1,R14,#@LB32
```

```
DO BXH,FROM=(R1,ArrayLast),
       TO=(R5,ArrayFirst),
       BY=(R4,=A(-EntryLen))
  JAS   R14,ProcessEntry
ENDDO ,
```

```
+          L      R1,ArrayLast
+          L      R5,ArrayFirst
+          L      R4,=A(-EntryLen)
+#@LB35    DC     0H
           JAS    R14,ProcessEntry
+#@LB36    DC     0H
+          BRXH   R1,R4,#@LB35
```

DO – Counting

```
LHI    R0,MaxItems
DO FROM=(R0)
  A      R14,0(,R1)
  LA     R1,4(,R1)
ENDDO ,
```

```
        LHI    R0,MaxItems
+#@LB20 DC     0H
        A      R14,0(,R1)
        LA     R1,4(,R1)
+#@LB21 DC     0H
+       BRCT   R0,#@LB20
```

```
DO FROM=(R0,MaxItems)
  A      R14,0(,R1)
  LA     R1,4(,R1)
ENDDO ,
```

```
+       L      R0,MaxItems
+#@LB23 DC     0H
        A      R14,0(,R1)
        LA     R1,4(,R1)
+#@LB24 DC     0H
+       BRCT   R0,#@LB23
```

DO – While and Until

```
DO WHILE=(CLI,0(R1),LE,C' ')
  AHI   R1,1
ENDDO ,
```

```
+        BRC    15,#@LB50
+#@LB51 DC     0H
         AHI    R1,1
+#@LB50 DC     0H
+        CLI    0(R1),C' '
+        BRC    13,#@LB51
```

```
DO UNTIL=(CLI,0(R1),GT,C' ')
  AHI   R1,1
ENDDO ,
```

```
+#@LB55 DC     0H
         AHI    R1,1
+#@LB56 DC     0H
+        CLI    0(R1),C' '
+        BRC    15-2,#@LB55
```

DO – Combining Other Keywords With While and/or Until

```
DO FROM=(R0),
   WHILE=(CLI,0(R1),LE,C' ')
  AHI   R1,1
ENDDO ,
```

```
+#@LB60 DC     0H
+       CLI    0(R1),C' '
+       BRC    15-13,#@LB59
        AHI    R1,1
+#@LB63 DC     0H
+       BRCT   R0,#@LB60
+#@LB59 DC     0H
```

```
DO WHILE=(CLI,0(R1),LE,C' '),
   UNTIL=(LTR,R15,R15,NZ)
  AHI   R1,1
  JAS   R14,ProcessChar
ENDDO ,
```

```
+#@LB65 DC     0H
+       CLI    0(R1),C' '
+       BRC    15-13,#@LB64
        AHI    R1,1
        JAS    R14,ProcessChar
+#@LB68 DC     0H
+       LTR    R15,R15
+       BRC    15-7,#@LB65
+#@LB64 DC     0H
```

DO – Demand Iteration

```
ITERATE [do_label]



OUTR DO INF,LABEL=OUTR
       JAS    R14,GetStmt
       DOEXIT LTR,R15,R15,NZ
       DO FROM=(R0)
         JAS    R14,ProcessKwd
         IF LTR,R15,R15,NZ
           ITERATE OUTR
         ENDIF ,
         AHI    R1,1
       ENDDO ,
       JAS    R14,PutResults
     ENDDO ,
```

```
+#@LB89 DC     0H
        JAS    R14,GetStmt
+       LTR    R15,R15
+       BRC    7,#@LB88
+#@LB93 DC     0H
        JAS    R14,ProcessKwd
+       LTR    R15,R15
+       BRC    15-7,#@LB95
+       BRC    15,#@LB89
+#@LB95 DC     0H
        AHI    R1,1
+#@LB94 DC     0H
+       BRCT   R0,#@LB93
        JAS    R14,PutResults
+       BRC    15,#@LB89
+#@LB88 DC     0H
```

DO – Demand Exit

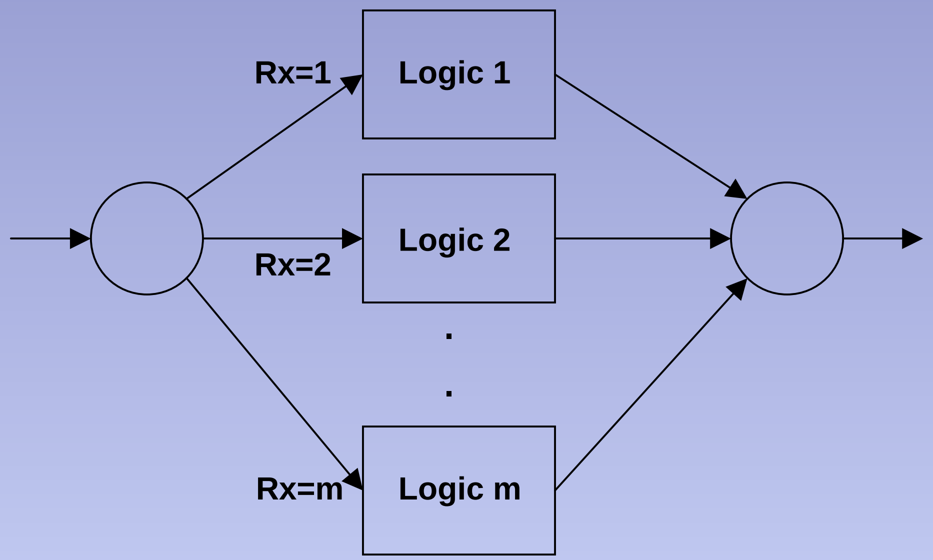
```
DOEXIT conditions[,DO=do_label]
ASMLEAVE [do_label]



OUTR DO UNTIL=(LTR,R15,R15,NZ)
       DO FROM=(R0)
         DOEXIT CLI,0(R1),GT,C' '
         JAS   R14,ProcessChar
         IF LTR,R15,R15,NZ
           MVI   FootPrint,C'C'
           ASMLEAVE OUTR
         ENDIF ,
         AHI   R1,1
       ENDDO ,
       JAS   R14,ProcessKwd
     ENDDO ,
```

```
+#@LB77 DC     0H
+#@LB82 DC     0H
+       CLI    0(R1),C' '
+       BRC    2,#@LB81
        JAS    R14,ProcessChar
+       LTR    R15,R15
+       BRC    15-7,#@LB86
        MVI    FootPrint,C'C'
+       BRC    15,#@LB76
+#@LB86 DC     0H
        AHI    R1,1
+#@LB83 DC     0H
+       BRCT   R0,#@LB82
+#@LB81 DC     0H
        JAS    R14,ProcessKwd
+       LTR    R15,R15
+       BRC    15-7,#@LB77
+#@LB76 DC     0H
```

DO – Alternate Labeling Method

```
ProcessKwds DO ,                   Do for keyword processing
         JAS    R14,GetNextKwd       Get next keyword
         .                           .
         ASMLEAVE ProcessKwds        Finished with keywords
         .                           .
         ITERATE ProcessKwds         Process next keyword
         .                           .
       ENDDO ,                     EndDo for keyword processing


       DO LABEL=ProcessKwds        Do for keyword processing
         JAS    R14,GetNextKwd       Get next keyword
         .                           .
         ASMLEAVE ProcessKwds        Finished with keywords
         .                           .
         ITERATE ProcessKwds         Process next keyword
         .                           .
       ENDDO ,                     EndDo for keyword processing
```

CASE Macro Set




```
CASENTRY Rx
CASE 1
  Logic 1
CASE 2
  Logic 2
.
.
CASE m
  Logic m
ENDCASE
```

Notes:

•Values in register x are powers of 2 (i.e., 1s, 2s, 4s, 8, 16s, etc.).

•Control passed via branch table. Very efficient for processing many uniformly distributed numeric values.

•Value of zero not supported (unfortunately).

•R0 destroyed when relative branch used.

CASE – Based Branch

```
CASENTRY R15
CASE 1
  BAS   R14,HandleCase1
CASE 2
  BAS   R14,HandleCase2
CASE 5
  BAS   R14,HandleCase5
ENDCASE ,
```

```
+         SLA   R15,2-0
+         A     R15,#@LB131
+         L     R15,0(,R15)
+         BCR   15,R15
+#@LB131 DC    A(#@LB129)
+#@LB132 DC    0H
          BAS   R14,HandleCase1
+         L     R15,#@LB129
+         BCR   15,R15
+#@LB133 DC    0H
          BAS   R14,HandleCase2
+         L     R15,#@LB129
+         BCR   15,R15
+#@LB134 DC    0H
          BAS   R14,HandleCase5
+         L     R15,#@LB129
+         BCR   15,R15
+#@LB129 DC    A(#@LB130)
+         DC    A(#@LB132)
+         DC    A(#@LB133)
+         DC    A(#@LB130)
+         DC    A(#@LB130)
+         DC    A(#@LB134)
+#@LB130 DC    0H
```

CASE – Relative Branch

```
CASENTRY R15
CASE 1
  JAS    R14,HandleCase1
CASE 2
  JAS    R14,HandleCase2
CASE 5
  JAS    R14,HandleCase5
ENDCASE ,
```

```
+         SLA    R15,2-0
+         LR     0,R15
+         CNOP   0,4
+         BRAS   R15,*+8
+         DC     A(#@LB118-*)
+         AL     R15,0(R15,0)
+         ALR    R15,0
+         BR     R15
+#@LB120  DC     0H
          JAS    R14,HandleCase1
+         BRC    15,#@LB119
+#@LB121  DC     0H
          JAS    R14,HandleCase2
+         BRC    15,#@LB119
+#@LB122  DC     0H
          JAS    R14,HandleCase5
+#@LB118  BRC    15,#@LB119
+         BRC    15,#@LB120
+         BRC    15,#@LB121
+         BRC    15,#@LB119
+         BRC    15,#@LB119
+         BRC    15,#@LB122
+#@LB119  DC     0H
```

Note: When **SYSSTATE ARCHLVL=2** is in effect, the blue fragment simplifies to:

```
+         LARL   0,#@LB118
```

CASE – Based Branch (Vector=B)

```
CASENTRY R15,POWER=2,VECTOR=B
CASE 4
  MVI   Severity,C'W'
CASE 8,12
  MVI   Severity,C'E'
CASE 16,20,24
  MVI   Severity,C'S'
ENDCASE ,
```

```
+          BC    15,#@LB108(R15)
+#@LB110 DC    0H
           MVI   Severity,C'W'
+          BC    15,#@LB109
+#@LB111 DC    0H
           MVI   Severity,C'E'
+          BC    15,#@LB109
+#@LB112 DC    0H
           MVI   Severity,C'S'
+#@LB108 BC    15,#@LB109
+          BC    15,#@LB110
+          BC    15,#@LB111
+          BC    15,#@LB111
+          BC    15,#@LB112
+          BC    15,#@LB112
+          BC    15,#@LB112
+#@LB109 DC    0H
```

CASE – Relative Branch (Vector=B)

```
CASENTRY R15,POWER=2,VECTOR=B
CASE 4
  MVI   Severity,C'W'
CASE 8,12
  MVI   Severity,C'E'
CASE 16,20,24
  MVI   Severity,C'S'
ENDCASE ,
```

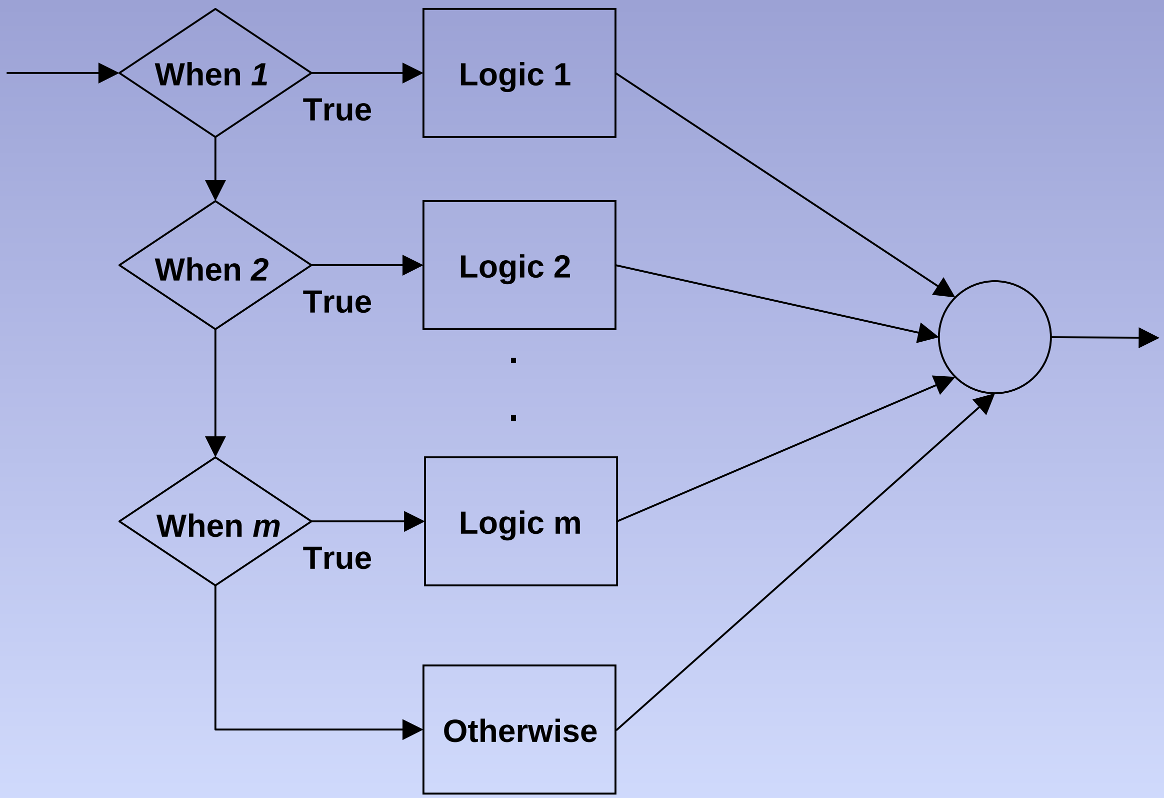
Note: The **VECTOR=** keyword is ignored for relative branch expansions.

Note: When **SYSSTATE ARCHLVL=2** is in effect, the blue fragment simplifies to:

```
+         LARL  0,#@LB123
```

```
+         LR    0,R15
+         CNOP  0,4
+         BRAS  R15,*+8
+         DC    A(#@LB123-*)
+         AL    R15,0(R15,0)
+         ALR   R15,0
+         BR    R15
+#@LB125 DC    0H
          MVI   Severity,C'W'
+         BRC   15,#@LB124
+#@LB126 DC    0H
          MVI   Severity,C'E'
+         BRC   15,#@LB124
+#@LB127 DC    0H
          MVI   Severity,C'S'
+#@LB123 BRC   15,#@LB124
+         BRC   15,#@LB125
+         BRC   15,#@LB126
+         BRC   15,#@LB126
+         BRC   15,#@LB127
+         BRC   15,#@LB127
+         BRC   15,#@LB127
+#@LB124 DC    0H
```

SELECT Macro Set



SELECT – Global Test

```
SELECT CLI,0(R1),EQ
WHEN C'A'
  LHI   R15,12
WHEN C'B'
  LHI   R15,16
WHEN C'C'
  LHI   R15,24
WHEN C'D'
  LHI   R15,8
OTHRWISE ,
  XR    R15,R15
ENDSEL ,
```

```
+          CLI    0(R1),C'A'
+          BRC    15-8,#@LB145
           LHI    R15,12
+          BRC    15,#@LB144
+#@LB145 DC     0H
+          CLI    0(R1),C'B'
+          BRC    15-8,#@LB147
           LHI    R15,16
+          BRC    15,#@LB144
+#@LB147 DC     0H
+          CLI    0(R1),C'C'
+          BRC    15-8,#@LB149
           LHI    R15,24
+          BRC    15,#@LB144
+#@LB149 DC     0H
+          CLI    0(R1),C'D'
+          BRC    15-8,#@LB151
           LHI    R15,8
+          BRC    15,#@LB144
+#@LB151 DC     0H
           XR     R15,R15
+#@LB144 DC     0H
```

SELECT – Unique Tests

```
SELECT ,
WHEN CLI,0(R1),EQ,0
  LHI   R15,12
WHEN CLI,0(R2),EQ,1
  LHI   R15,16
WHEN CLI,0(R3),EQ,2
  LHI   R15,24
WHEN CLI,0(R4),EQ,9
  LHI   R15,8
OTHRWISE ,
  XR    R15,R15
ENDSEL ,
```

```
+        CLI   0(R1),0
+        BRC   15-8,#@LB136
         LHI   R15,12
+        BRC   15,#@LB135
+#@LB136 DC    0H
+        CLI   0(R2),1
+        BRC   15-8,#@LB138
         LHI   R15,16
+        BRC   15,#@LB135
+#@LB138 DC    0H
+        CLI   0(R3),2
+        BRC   15-8,#@LB140
         LHI   R15,24
+        BRC   15,#@LB135
+#@LB140 DC    0H
+        CLI   0(R4),9
+        BRC   15-8,#@LB142
         LHI   R15,8
+        BRC   15,#@LB135
+#@LB142 DC    0H
         XR    R15,R15
+#@LB135 DC    0H
```

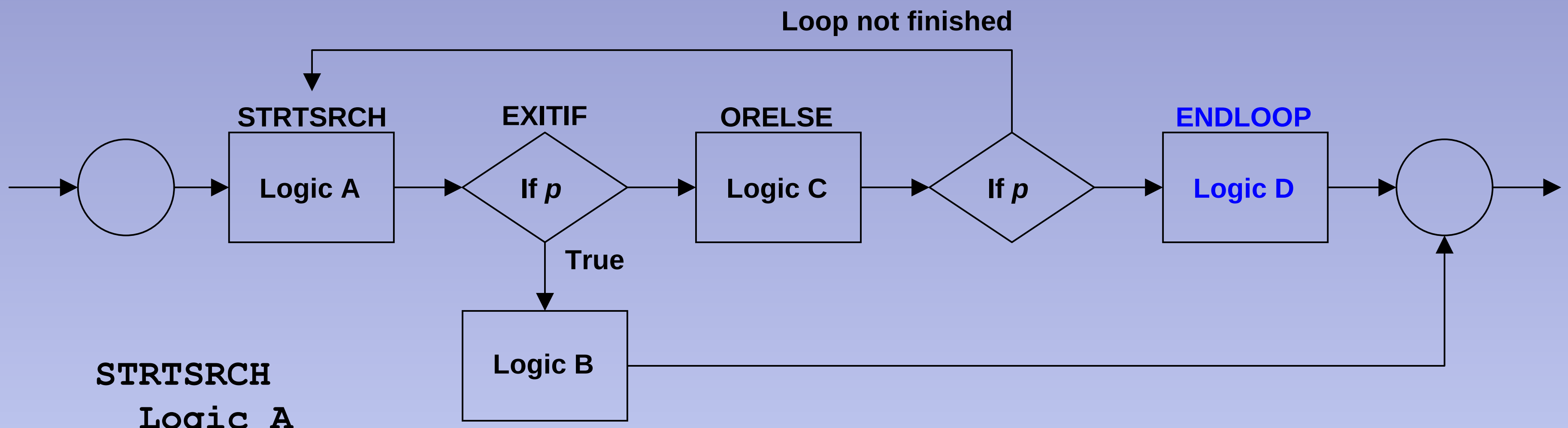
Defeating the Mutual-Exclusivity of the WHEN Clause

- WHEN clauses are always mutually exclusive. This can lead to duplicated logic.
- One of my enhancements adds NEXTWHEN. When encountered, it passes control to the next WHEN (or OTHRWISE) clause.
- NEXTWHEN may appear anywhere within a WHEN clause (even from inside other constructs such as IF or DO).

```
SELECT ,
WHEN CLI,0(R1),EQ,0
  OI    FLAG1,Zero
  OI    FLAG2,SingleDigit
WHEN CLI,0(R1),LT,10
  OI    FLAG2,SingleDigit
ENDSEL ,
```

```
SELECT ,
WHEN CLI,0(R1),EQ,0
  OI    FLAG1,Zero
  NEXTWHEN ,
WHEN CLI,0(R1),LT,10
  OI    FLAG2,SingleDigit
ENDSEL ,
```

SEARCH Macro Set




```
STRTSRCH
  Logic A
EXITIF (p)
  Logic B
ORELSE
  Logic C
ENDLOOP
  Logic D
ENDSRCH
```

Notes:
- STRTSRCH has same loop control options as DO.
- ENDLOOP (Logic D) differentiates SEARCH from DO.
- DOEXIT and ASMLEAVE go to ENDLOOP logic.
- EXITIF and ORELSE are optional.
- Each EXITIF (except the last) must be followed by an ORELSE.

Why I Never Use SEARCH

- Any mature product has obsolete commands/features. They tend to be created to fix a specific problem. Later, that same problem is addressed in a more generalized way and the “stop-gap” solution becomes obsolete.
- At one time SEARCH was necessary to address deficiencies in the more general DO macro set.
  - No simple DO.
  - No DOEXIT support for compound tests.
  - No DOEXIT/ASMLEAVE from inner constructs or nested DOs.
  - These and other similar deficiencies have all been resolved.
- SEARCH has no direct counterpart in other structured languages, making it undesirable for general-purpose use.

THE END