Assembler Concepts

An Old, Durable Architecture

- System 360 (1964)
- System 370 (1971)
- 370 Extended Architecture –XA (1983)
- Enterprise Systems Architecture – ESA (1989)
- ESA 390 (1990-1999)
- Z/Architecture (2000)

Assembler Documentation

- The two most important assembler manuals are
  - Principles of Operation - describes how the machine and each instruction works
  - Language Reference – describes how to use the assembler directives

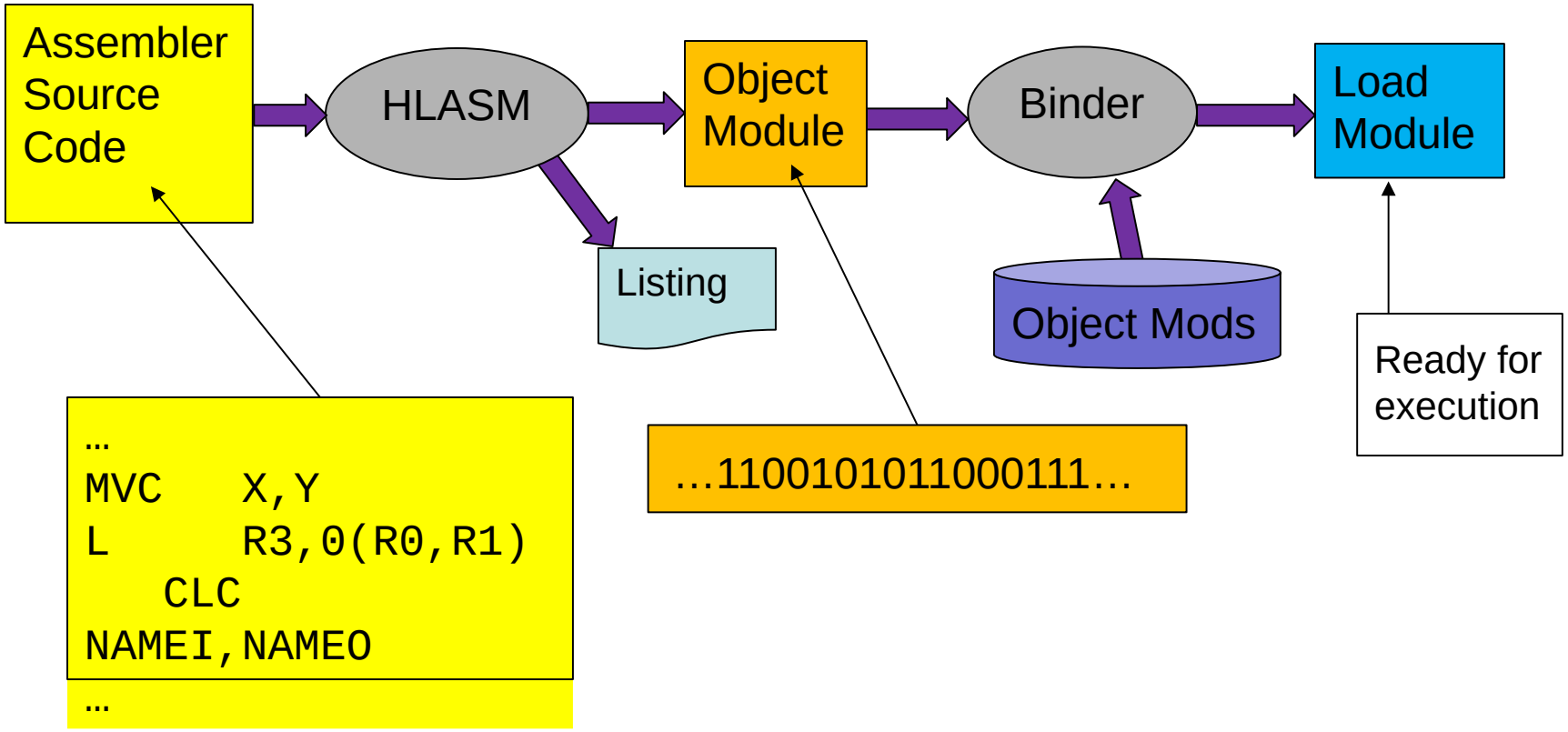

IBM System/360 Reference Data

MACHINE INSTRUCTIONS

Documentation

- IBM documentation for the High-level Assembler (HLASM) is available over the web:
- Principles of Operations
  - SA22-7832-10 http://publibfp.dhe.ibm.com/epubs/pdf/dz9zr010.pdf
- High Level Assembler Language Reference
  - SC26-4940-07 http://publibfp.dhe.ibm.com/epubs/pdf/asmr1022.pdf

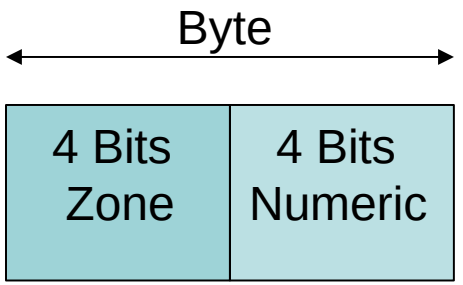
Translation of Assembler Code



Architecture - Memory

- **Bit**
  - Binary Digit (1 or 0)

- **Byte**
  - Zone Portion – 4 Bits
  - Numeric Portion – 4 Bits



- Bytes are numbered (each byte has an address) starting at 0 and getting larger

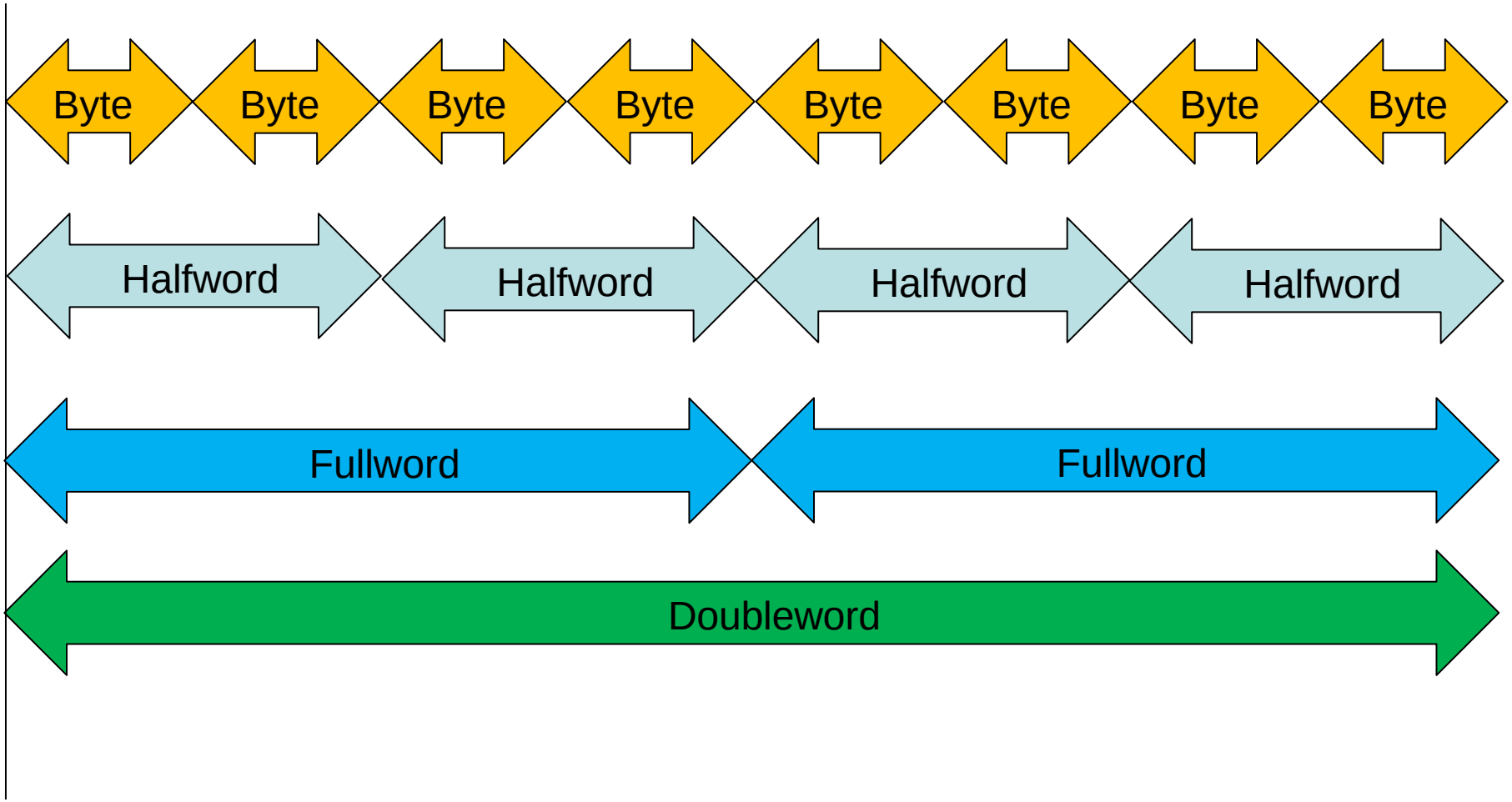
Address 0 1 2 3 4 …

- Bytes are the smallest storage unit with an address

Architecture - Memory

- **Halfword**
  - 2 Bytes
  - Halfword boundary (Address evenly divisible by 2)
- **Fullword**
  - 4 Bytes
  - Word Boundary (Address evenly divisible by 4)
- **Doubleword**
  - 8 Bytes
  - Double Word boundary (Address evenly divisible by 8)
- **Quadword**
  - 16 Bytes
  - Quad Word boundary (Address evenly divisible by 16)

Architecture - Memory



Terms that Describe Memory

Metric

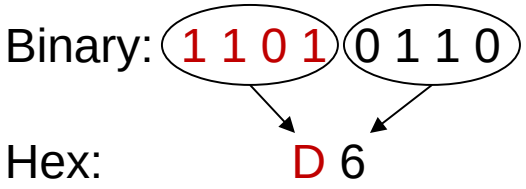
1000	KB	kilobyte
1000^2	MB	megabyte
1000^3	GB	gigabyte
1000^4	TB	terabyte
1000^5	PB	petabyte
1000^6	EB	exabyte
1000^7	ZB	zettabyte
1000^8	YB	yottabyte

Terms that Describe Memory

			IEC	JEDEC
$1024 = 2^{10}$	KiB	kibibyte	KB	kilobyte
$1024^2 = 2^{20}$	MiB	mebibyte	MB	megabyte
$1024^3 = 2^{30}$	GiB	gibibyte	GB	gigabyte
$1024^4 = 2^{40}$	TiB	tebibyte	–	
$1024^5 = 2^{50}$	PiB	pebibyte	–	
$1024^6 = 2^{60}$	EiB	exbibyte	–	
$1024^7 = 2^{70}$	ZiB	zebibyte	–	
$1024^8 = 2^{30}$	YiB	yobibyte	–	

Representing Memory

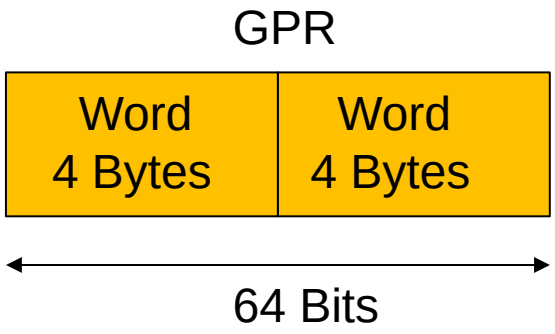
- The contents of memory is often displayed in hexadecimal (base 16) digits
- 1 hex digit = 4 bits
- 2 hex digits = 1 byte



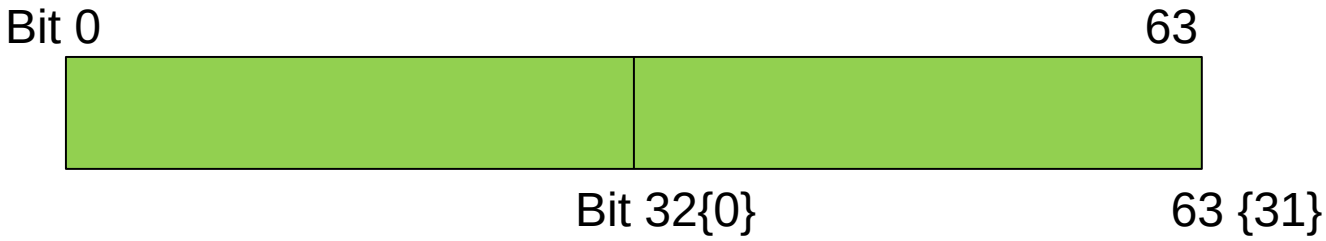
Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Architecture - Registers

- 16 General Purpose Registers
  - 64 Bits in a Register
  - 8 Bytes in a Register
  - 2 Fullwords in a Register
  - Used for Binary Arithmetic
  - Used for creating Addresses
  - Numbered 0 – 15
  - Rightmost word functions as a 32-bit register



64-Bit GP Registers



- Bits in a register are numbered 0 to 63 moving from left to right
- The rightmost 32 bits can function as a 32-bit register. In this case, we number those bits 32-63 or 0-31 depending on the context
- In both cases, the register number is the same

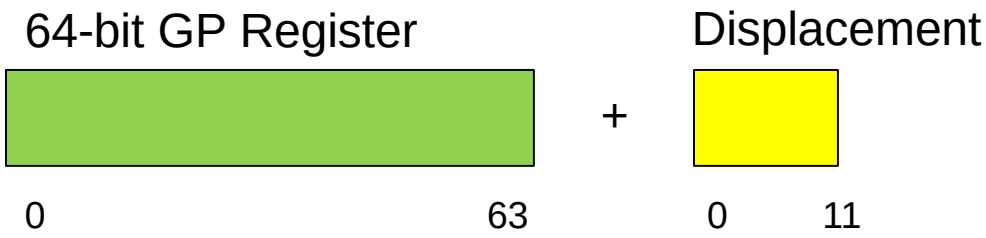
Addressing

- Addressing is the process of specifying the location of a byte in memory
- Addressing varies based on instruction type
  - Base Register + Displacement
  - Base Register + Index Register + Displacement
  - Relative

Addressing

- Base/Displacement Scheme
  - Makes Object code smaller
  - Makes Linkage simpler
- A specific byte is indicated by specifying a beginning address (base) and an offset from the base (displacement)
- Base address contained in a General Purpose Register
- Displacement specified in an instruction

Base/Displacement Addressing



- The contents of the base register are added to the displacement
- The result is truncated on the left depending on the machine's current addressing mode
  - In 24-bit mode, the leftmost 40 bits are set to zeros.
  - In 31-bit mode, the leftmost 33 bits are set to zeros.
  - In 64-bit mode, the 64-bit address is not truncated.
- Initially we will use 31 bit addressing

Addressing

In an assembled instruction, a Base/Displacement address looks like 4 hex digits:

BDDD

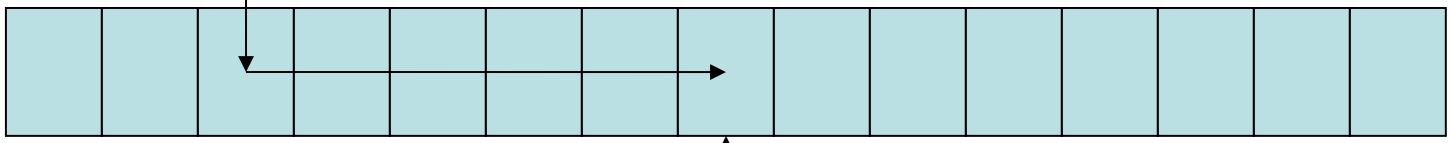
B = Base Register

DDD = Displacement (0 – 4095)

Register 10

1500

Memory



Base Displacement Address:

A005

1505

Object Code

How a programmer codes it

In explicit notation an address looks like D(B) or

D(X, B)

D is a displacement

B is a base register

X is an index register

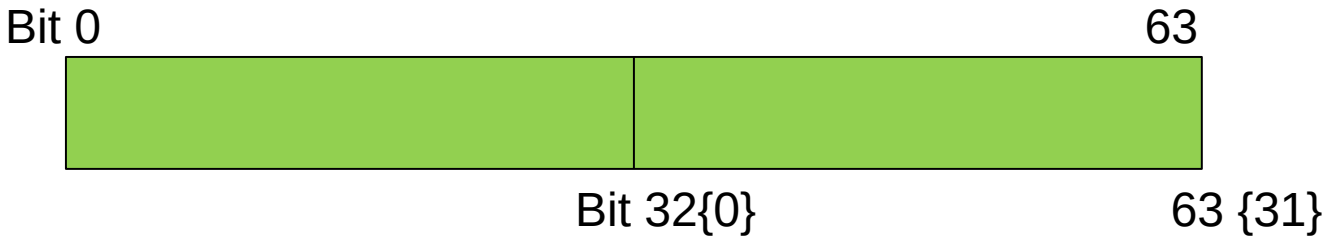
Trimodal Addressing

- The machine supports three addressing modes (24, 31, and 64)
- Each mode determines the number of bits used to make up an address
- 24 bits address $2^{24} = 2^4 \times 2^{20} = 16$ MB
- 31 bits can address $2^{31} = 2^1 \times 2^{30} = 2$ GB
- 64 bits can address $2^{64} = 2^4 \times 2^{60} = 16$ EB
- 5 Exabytes: All words ever spoken by human beings.

Modal vs Non-Modal

- A **modal** instruction is one that is affected when you change addressing modes
- A **non-modal** instruction is one that is not affected by changing addressing modes

Using 64-Bit GP Registers



- You don’t have to be running in 64-bit addressing mode to use a 64-bit register
- A program can change addressing modes while it runs

In the First Course

- Registers are 32 bits (rightmost bits)
- We are always running in 31-bit addressing

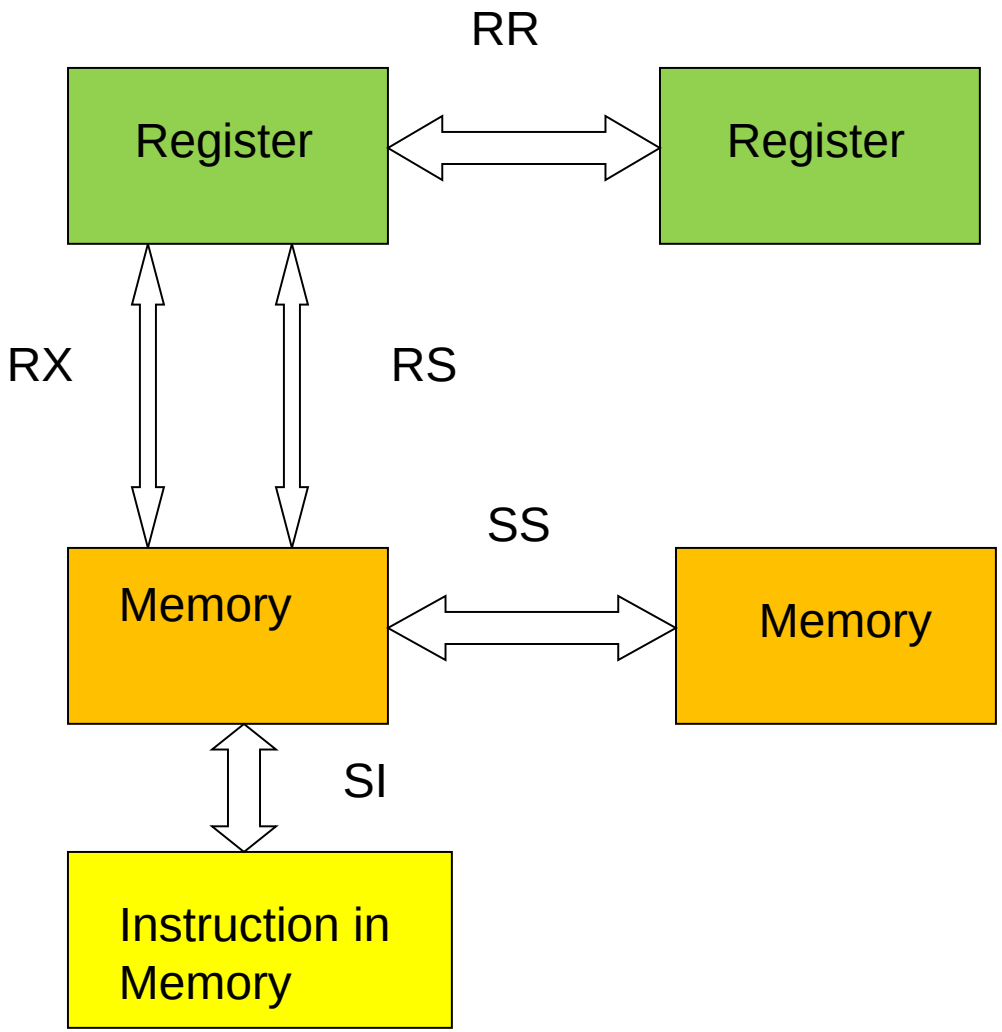
Instructions

- There are well over 1000 instructions on the latest IBM System/z machine!
- Many have very specific uses
- Some instructions are “privileged” and require special permission to execute
- We are studying “non-privileged” instructions
- We will target a rich working subset of instructions

Instruction Types

- Instructions are categorized by type
- Types control the format of an instruction
- Learning about a type will help you learn characteristics of every instruction of that type
- We concentrate on five types: RR, RX, RS, SS, and SI
- There are many more types!
- Most instructions involve two operands

Location of Data



Architecture – Program Status Word

- PSW is a logical collection of data that indicates the current status of the machine
- Contains two important fields for programmers:
  - Condition Code (2 bits)
  - Instruction Address (24, 31, or 64 bit addresses)

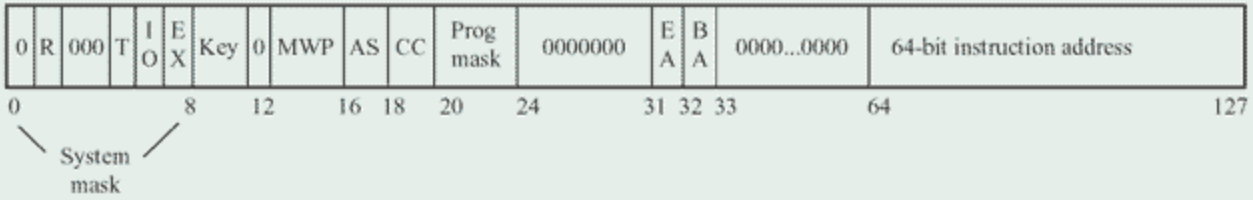


Figure 7

PSW format for z/Architecture.

Architecture – Condition Code

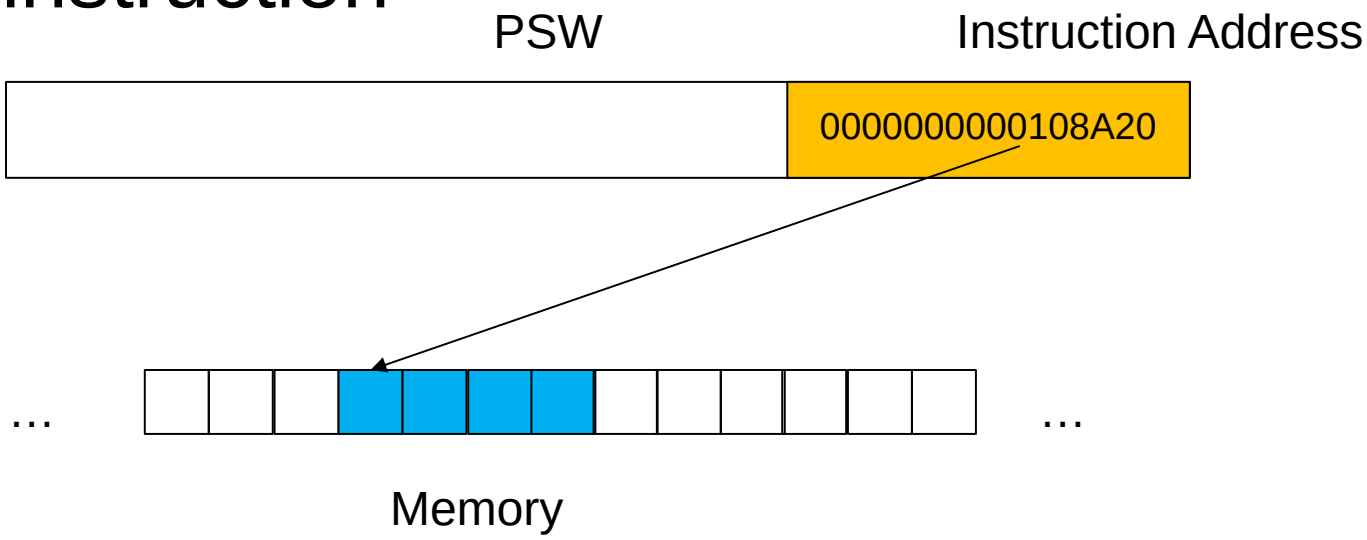
- Condition code (2 bits)
  - 00 – equal
  - 01 – low
  - 10 – high
  - 11 – overflow

- Test the condition code with Branch Instructions

```
CLC     X,Y             SET COND CODE
BE      THERE           TEST CC
...
THERE   EQU   *
```

Architecture – Instruction Address

- Contains the address of the “next” instruction

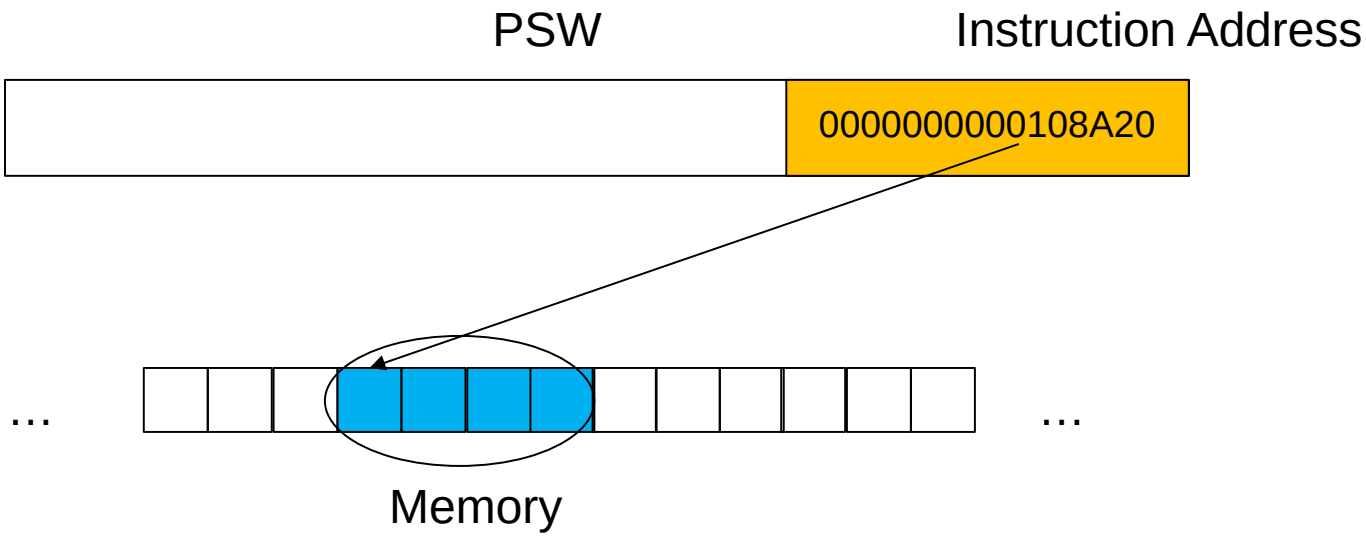


How does a computer work?

- **Fetch/Execute Cycle**
  - 1) Fetch the next instruction
  - 2) Decode the instruction
  - 3) Update the Instruction address field
  - 4) Execute the decoded instruction
  - 5) Go to step 1

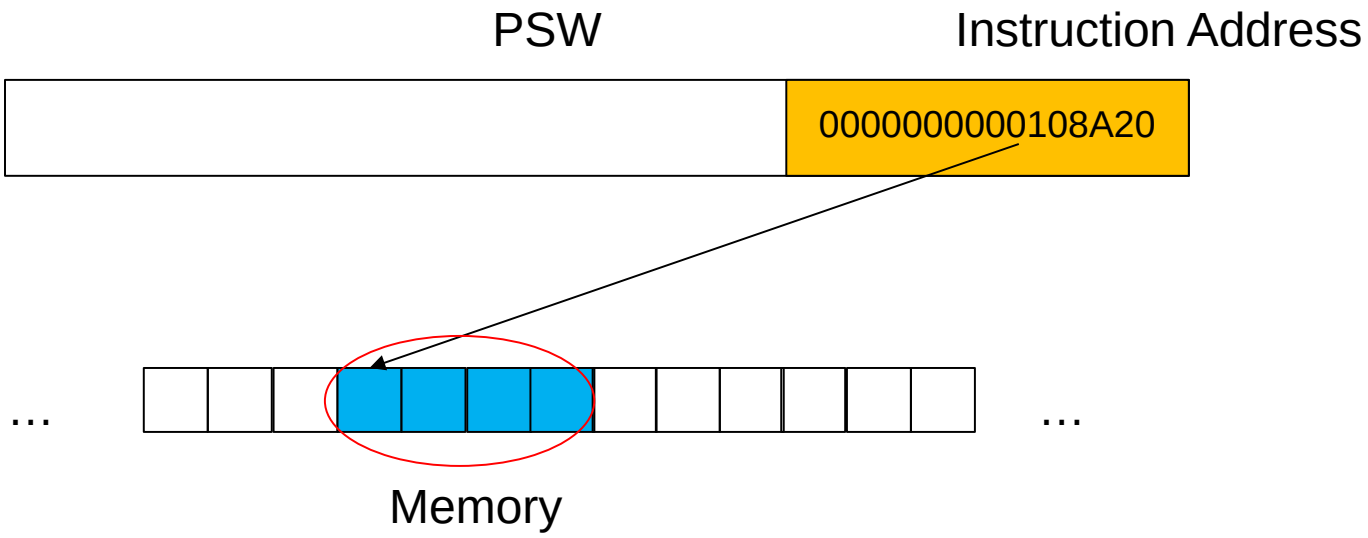
Architecture – Instruction Address

Fetch



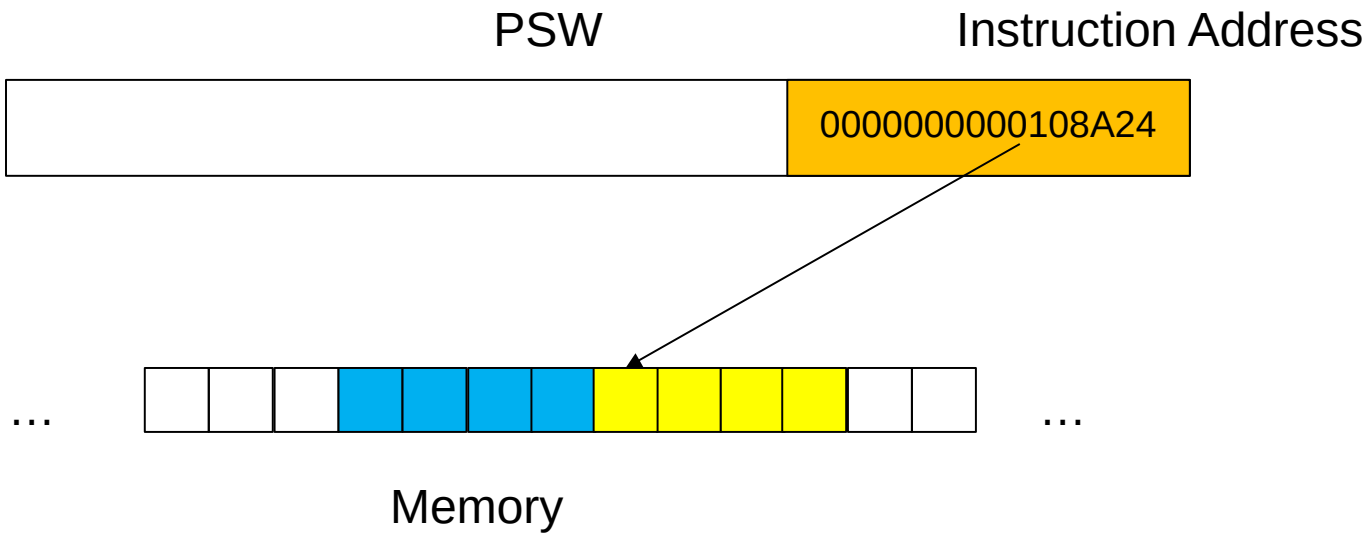
Architecture – Instruction Address

Decode



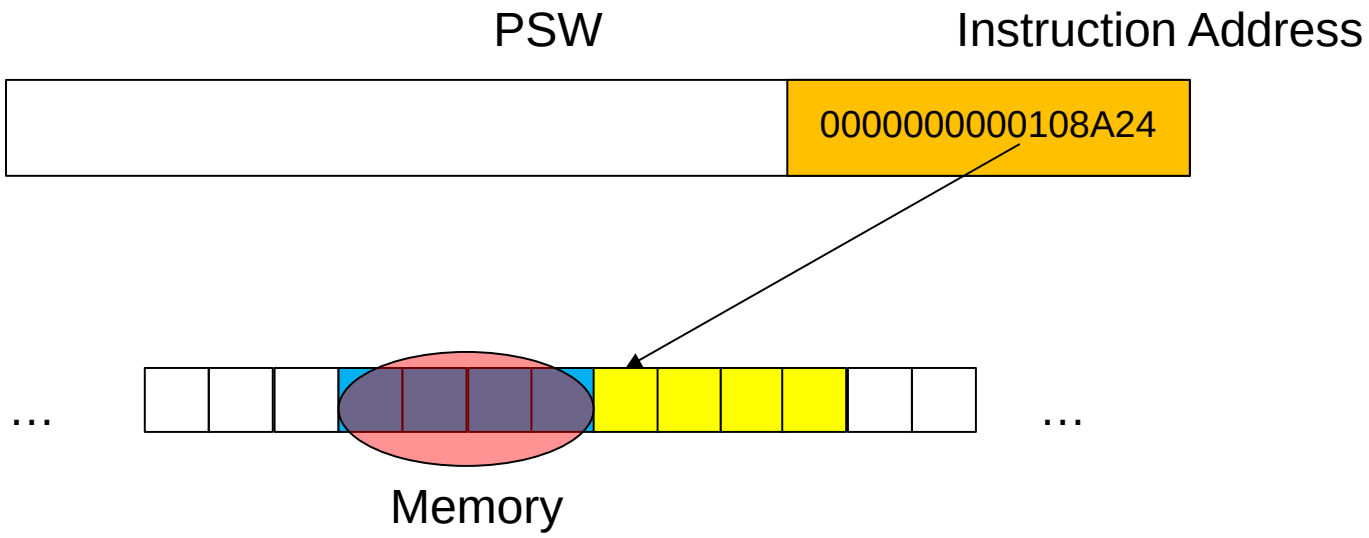
Architecture – Instruction Address

Update Instruction Address



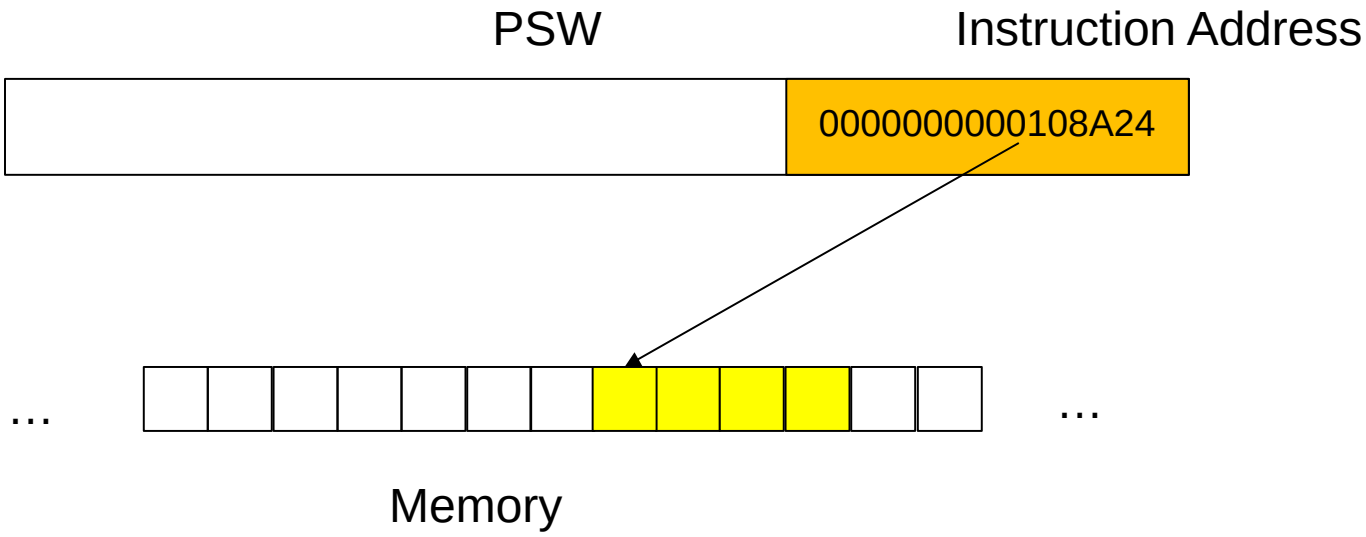
Architecture – Instruction Address

Executes Decoded Instruction



Architecture – Instruction Address

Go Back to Step 1



How does a computer work?

- **Fetch/Execute Cycle**
  - 1) Fetch the next instruction
  - 2) Decode the instruction
  - 3) Update the Instruction address field
  - 4) Execute the decoded instruction
  - 5) Go to step 1
- How does branching work?

A Standard Line of Assembler

- Assembler statements are usually coded on one line (80 characters)
- Longer statements can be continued

Columns 1 – 71 form a statement field	72	73 - 80

Column 72 – Continuation (any non-blank character)
Columns 73-80 – Identification sequence field – can be left blank
Column 16 – the continue column where continued lines start

EBCDIC Encoding

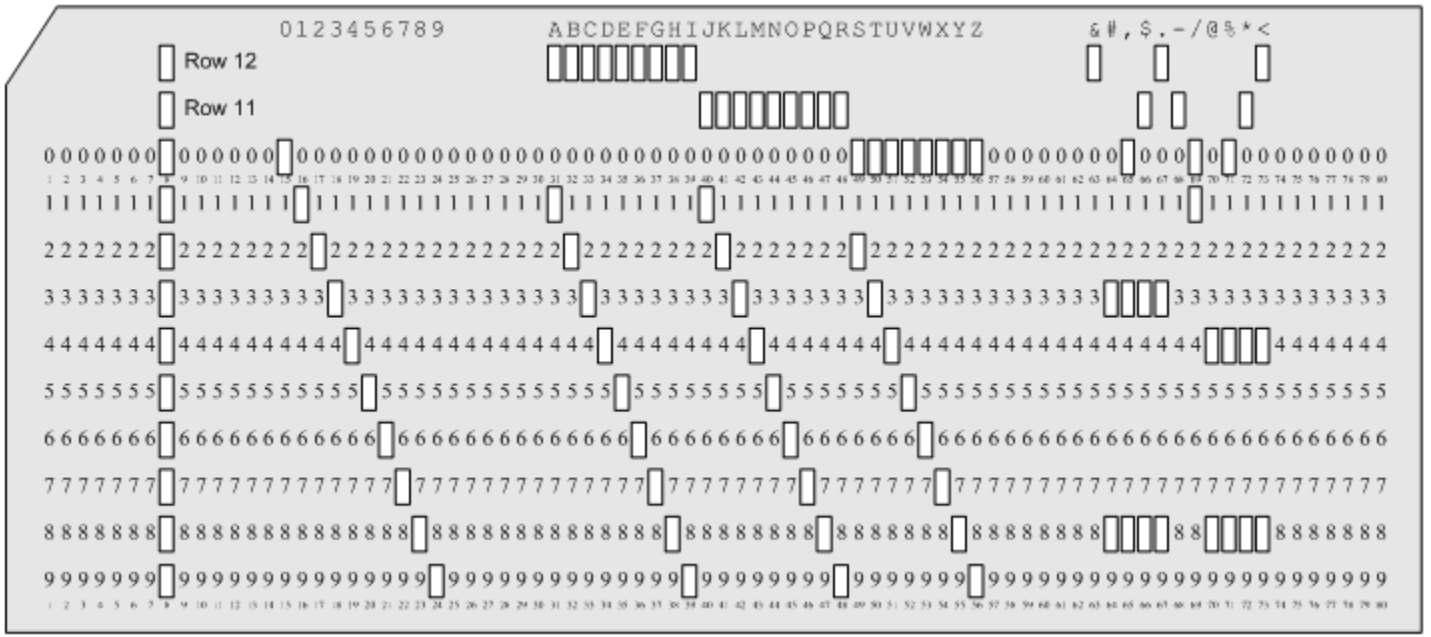
Character	Hex Equivalent
A	C1
B	C2
C	C3
D	C4
E	C5
F	C6
G	C7
H	C8
I	C9
J	D1
K	D2
L	D3
M	D4
N	D5
O	D6
P	D7
Q	D8
R	D9

Character	Hex Equivalent
S	E2
T	E3
U	E4
V	E5
W	E6
X	E7
Y	E8
Z	E9
BLANK	40
COMMA	6B
PERIOD	4B
ASTERISK	5C

EBCDIC Encoding

Character	Hex Equivalent
0	F0
1	F1
2	F2
3	F3
4	F4
5	F5
6	F6
7	F7
8	F8
9	F9

EBCDIC Encoding



"The herein described method of compiling statistics which consists in recording separate statistical items pertaining to the individual by holes or combinations of holed punched in sheets of electrically non-conducting material, and bearing a specific relation to each other and to a standard, and then counting or tallying such statistical items separately or in combination by means of mechanical counters operated by electro-magnets the circuits through which are controlled by the perforated sheets, substantially as and for the purpose set forth."

Defining Data and Data Areas

- Format for defining data

```
name        DS    rTLn
name        DC    rTLn'constant'
```

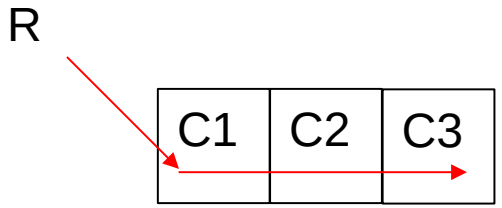
- r – repetition factor
- T – data type
- L – length
- n - integer

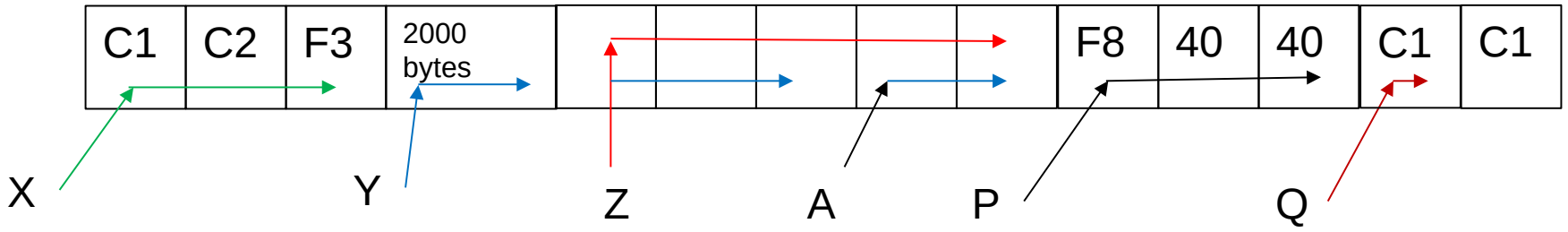
Character Data

- One character per byte
- EBCDIC representation (hex digits)
- Max DC size is 256
- Max DS size is 65,535
- Express constant as a character string in single quotes
- Constants are left justified and padded with blanks on the right or truncated on the right
- Symbols refer to the first byte of the field and have a length attribute

Data Types - Character

```
X       DC    CL3'AB3'
Y       DS    CL2000
Z       DS   0CL5
        DS    CL3
A       DS    CL2
P       DC    CL3'8'
Q       DC   2C'A'
R       DC    CL3'ABCD'
```





Location Counter

- As statements are assembled, the assembler keeps up with locations of identifiers by maintaining a location counter

```
  LOC
00000000 W        DS   CL3
00000003 X        DS   CL4
00000007 Y        DS   CL8
0000000F Z        DS   CL4
00000013
```

Location Counter

- A zero duplication factor leaves the location counter in the same position

```
  LOC
00000000 W        DS    CL3
00000003 X        DS    CL4
00000007 NAME     DS   0CL30
00000007 FNAME    DS    CL10
00000011 LNAME    DS    CL20
00000025          …
```

Location Counter

The location counter can be set with an orginate (ORG) assembler directive

```
  LOC
00000000 W        DS    CL3
00000003 X        DS    CL4
00000007 NAME     DS    CL30
00000025          ORG   NAME
00000007 FNAME    DS    CL10
00000011 LNAME    DS    CL20
00000025          …
```

Building Reports with ORG

```
 LOC
00001000 RPTLINE DC   CL80' '
00001050         ORG  RPTLINE+10
0000100A MON     DC   CL3'MON'
0000100D         ORG  RPTLINE+20
00001014 TUE     DC   CL3'TUE'
00001017         ORG  RPTLINE+30
0000101E WED     DC   CL3'WED'
00001021         ORG  RPTLINE+40
00001028 THU     DC   CL3'THU'
0000102B         ORG  RPTLINE+50
00001032 FRI     DC   CL3'FRI'
00001035         ORG  ,
00001050         ...
```

Resets LOC to RPTLINE+20 = 1014

Comma indicates no operand. Location counter set to highest unassigned location. A common error is to use ORG and leave the location counter in the middle of assigned storage

Character Instructions - MVC

- Move Characters
- SS_1
- Op1 – target field
- Op2 – source field
- Length – associated with operand 1 only!
- Default Length taken from operand 1
- Fields may overlap
- Max 256 bytes are moved (copied)

```
MVC  X,Y
MVC  X(2),Y
```

```
... X   DS  CL3
    Y   DS  CL3
```

SS_1 Object Instruction Format

- Storage to Storage (SS_1)

  General Format:

OP	LL_1	B_1D_1	D_1D_1	B_2D_2	D_2D_2

- OP – Operation code
- LL_1 – Length of Op One – 1 (Max 255)
- $B_1D_1D_1D_1$ – Base/Disp of Operand 1
- $B_2D_2D_2D_2$ – Base/Disp of Operand 2

Three ways to Code Instructions

- MVC FNAMEO,FNAMEI
  Symbolic addresses
- MVC 4(3,12),8(12) Explicit addresses
- MVC X,4(8) Mixed addresses
- MVC 0(3,R4),4(R9) Explicit addresses with equate symbols used for registers

  R4 EQU 4

Explicit Instruction Format

- Storage to Storage (SS_1)

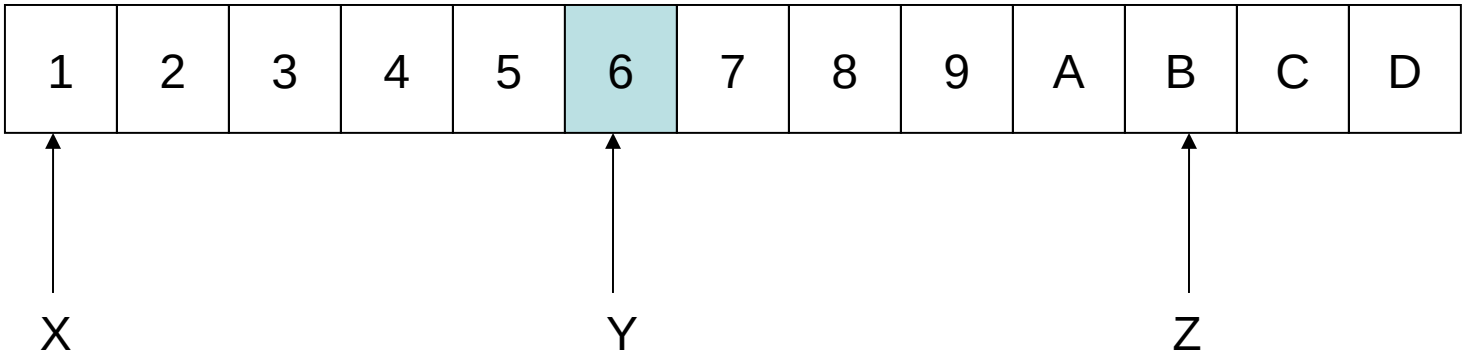
  Explicit Instruction format

  $OP \quad D_1(L_1,B_1),D_2(B_2)$

- OP – Operation code
- D_1, D_2 – Displacement of operand 1 and 2
- B_1, B_2 – Base regs of operand 1 and 2
- L_1 – Length of operand 1- (1- 256 decimal)

MVC EXAMPLE 1

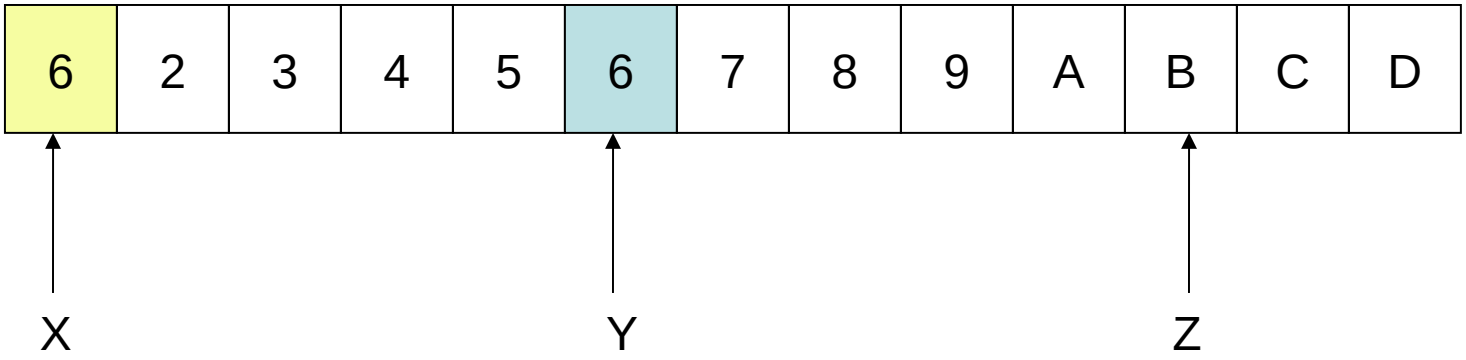
```
X       DC    CL5’12345’
Y       DC    CL5’6789A’
Z       DC    CL3’BCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 1

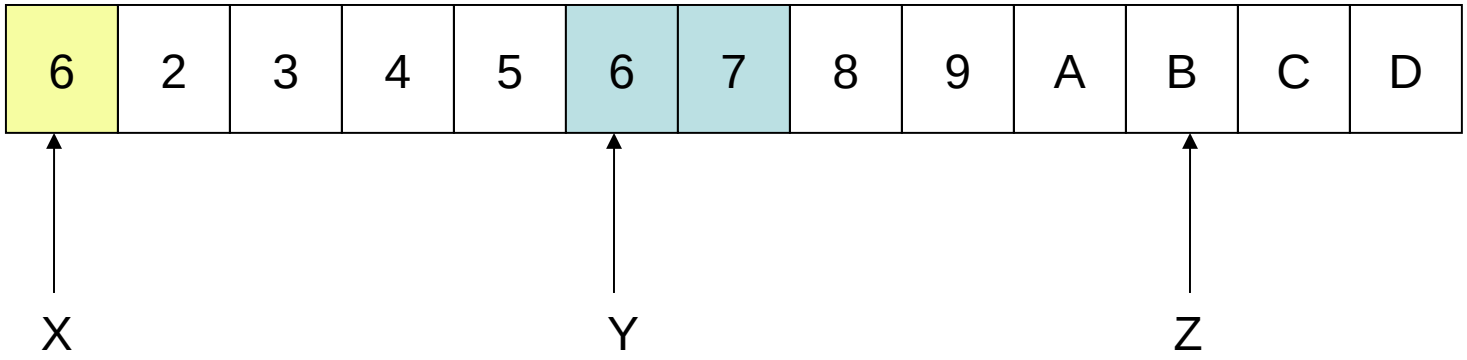
```
X       DC    CL5’12345’
Y       DC    CL5’6789A’
Z       DC    CL3’BCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 1

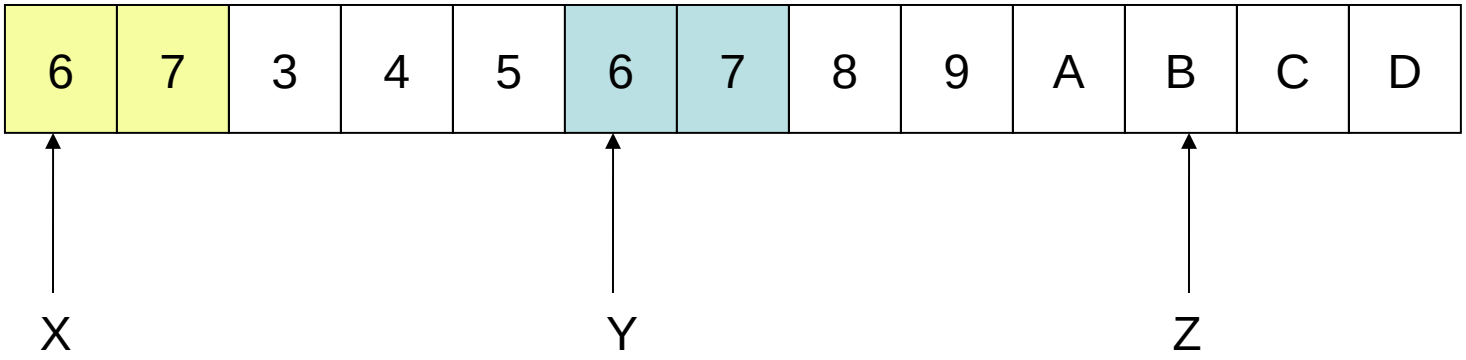
```
X       DC    CL5’12345’
Y       DC    CL5’6789A’
Z       DC    CL3’BCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 1

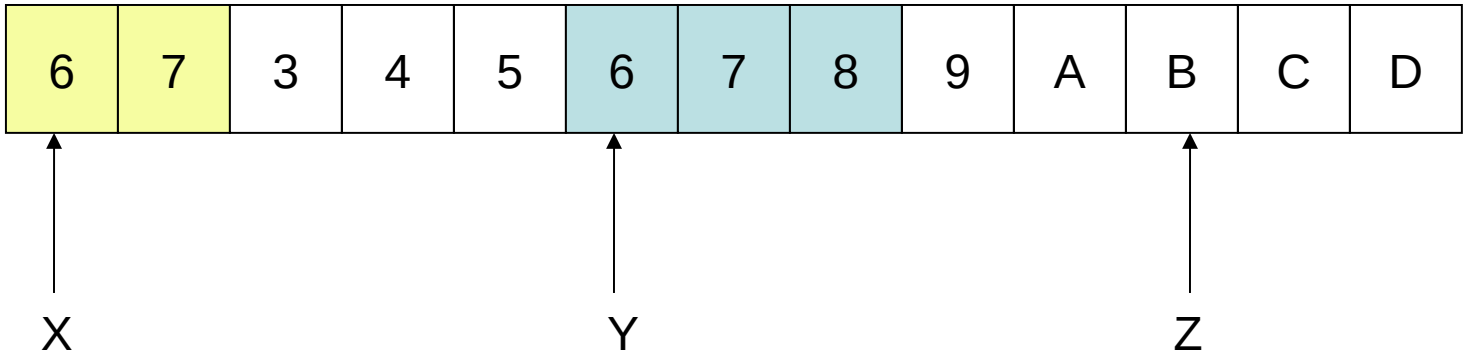
```
X       DC    CL5’12345’
Y       DC    CL5’6789A’
Z       DC    CL3’BCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 1

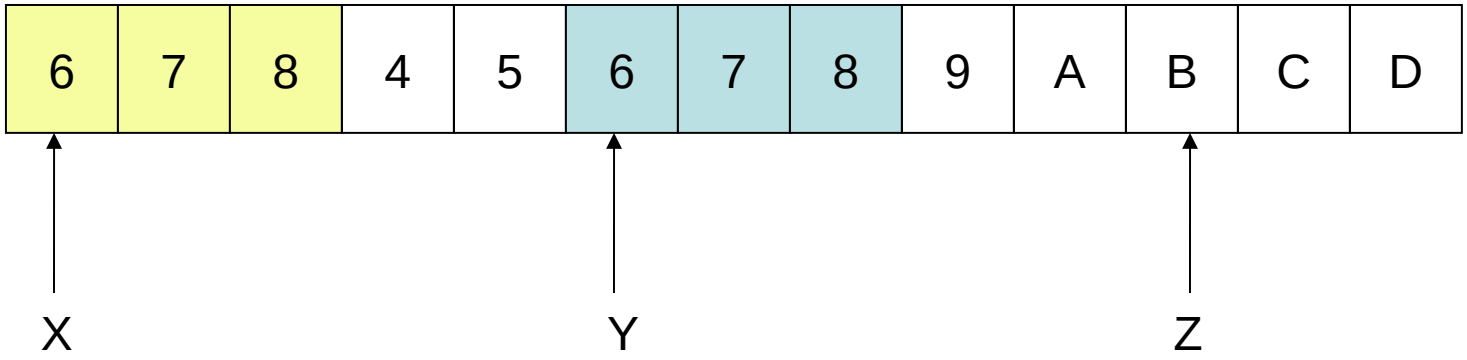
```
X       DC    CL5’12345’
Y       DC    CL5’6789A’
Z       DC    CL3’BCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 1

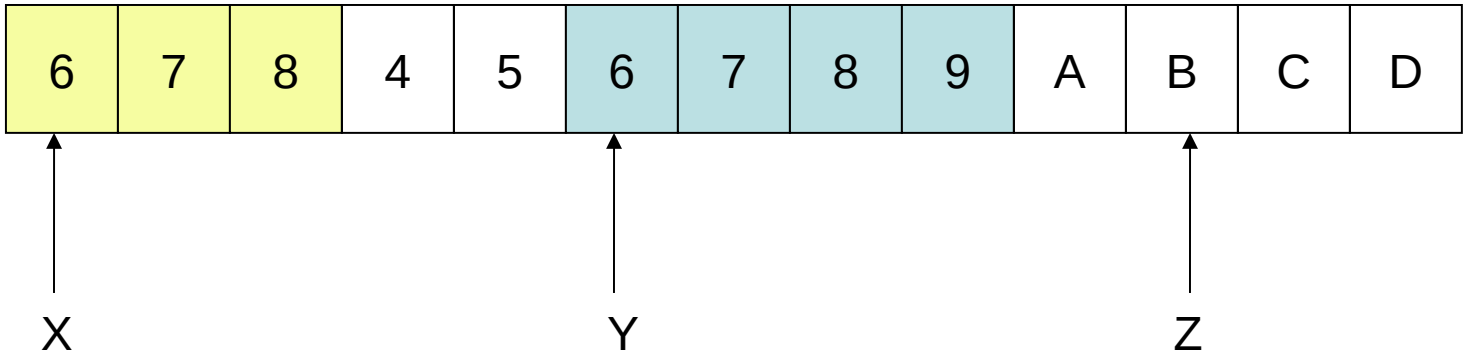
```
X       DC    CL5’12345’
Y       DC    CL5’6789A’
Z       DC    CL3’BCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 1

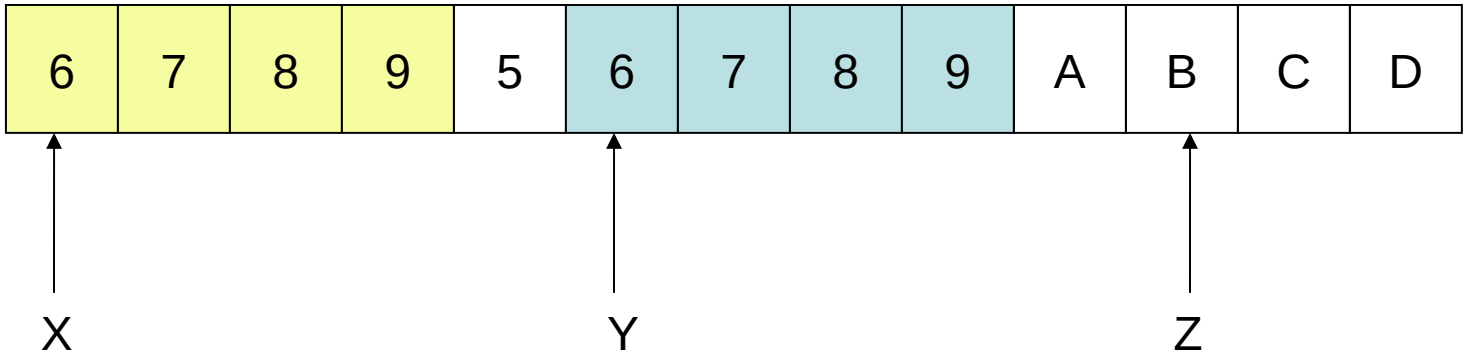
```
X       DC    CL5’12345’
Y       DC    CL5’6789A’
Z       DC    CL3’BCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 1

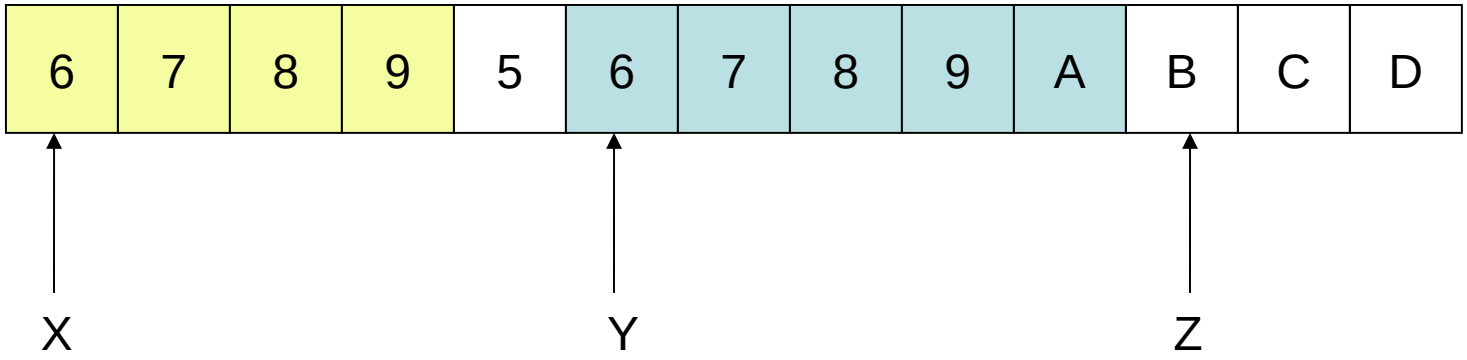
```
X       DC    CL5’12345’
Y       DC    CL5’6789A’
Z       DC    CL3’BCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 1

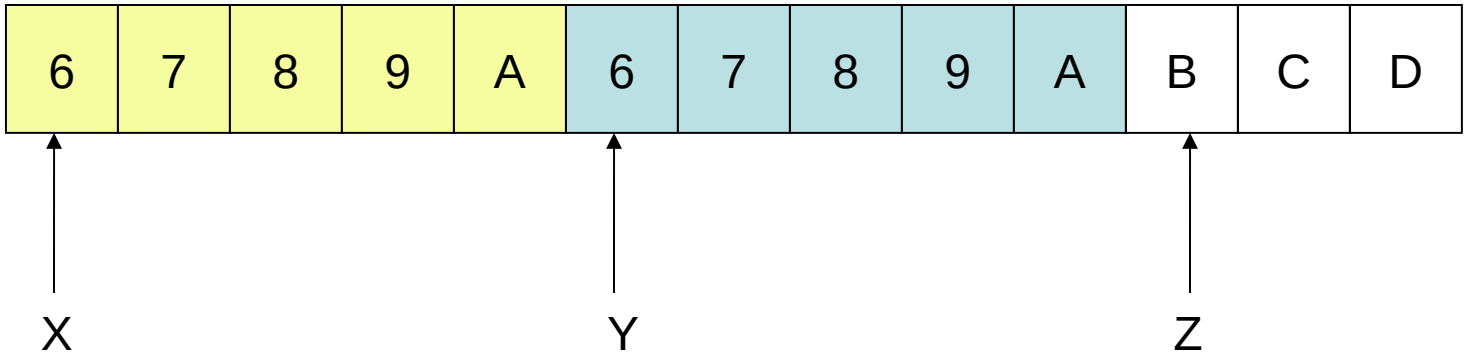
```
X       DC    CL5’12345’
Y       DC    CL5’6789A’
Z       DC    CL3’BCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 1

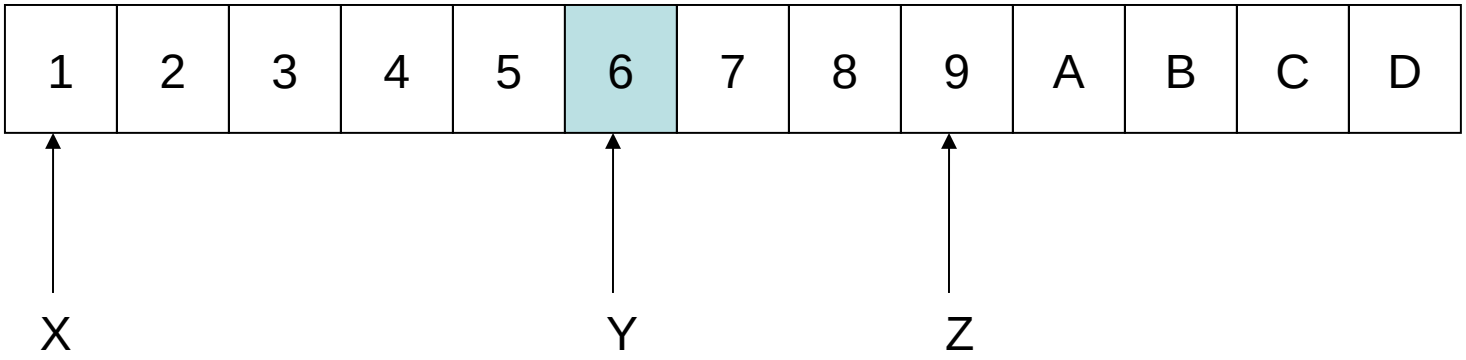
```
X       DC    CL5’12345’
Y       DC    CL5’6789A’
Z       DC    CL3’BCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 2

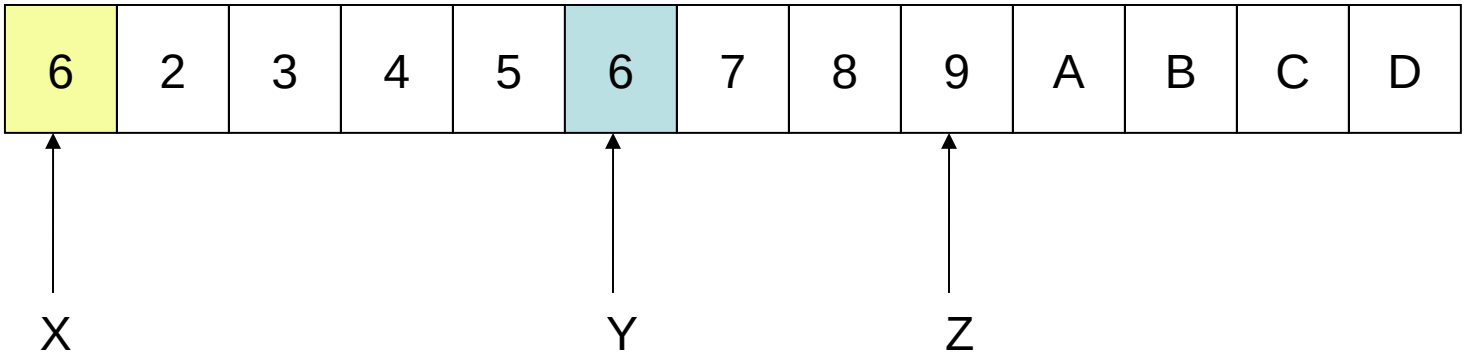
```
X       DC    CL5’12345’
Y       DC    CL3’678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 2

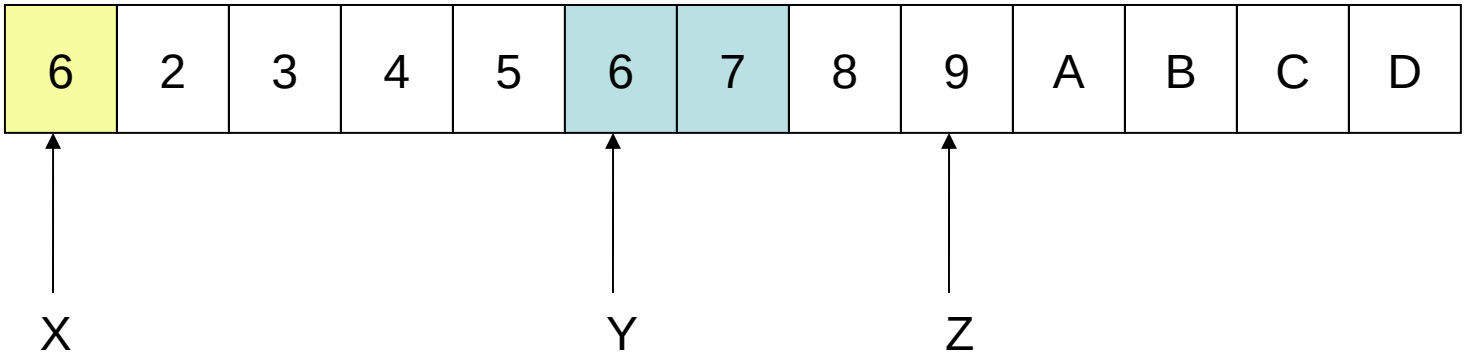
```
X       DC    CL5’12345’
Y       DC    CL3’678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 2

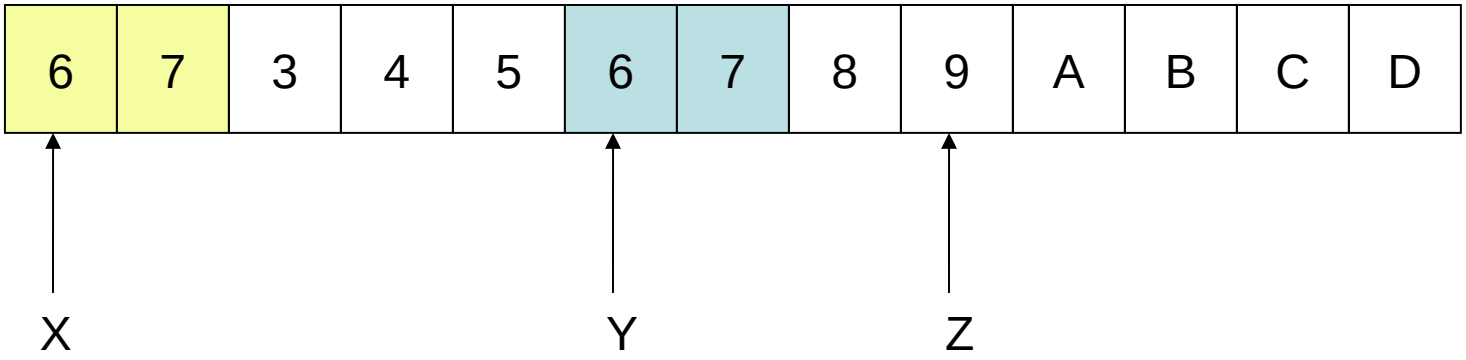
```
X       DC    CL5’12345’
Y       DC    CL3’678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 2

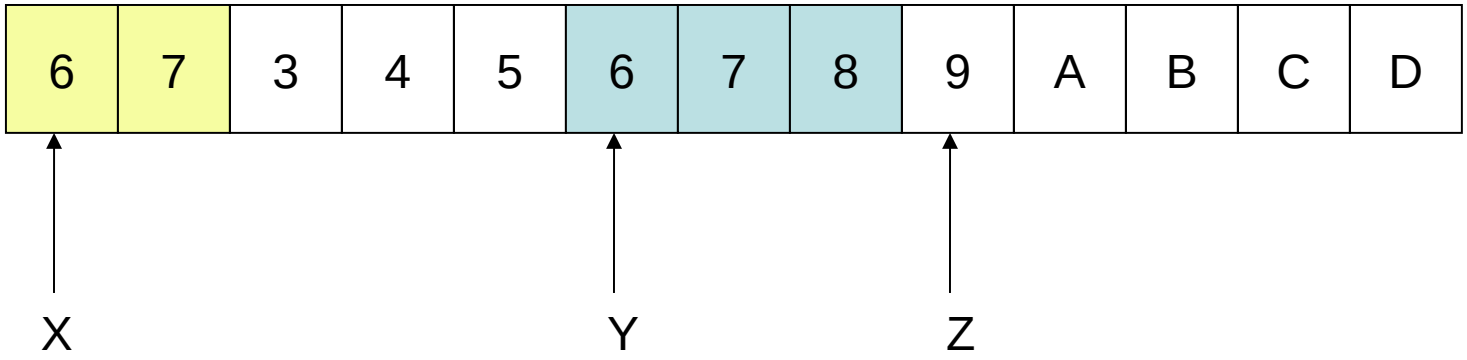
```
X       DC    CL5’12345’
Y       DC    CL3’678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 2

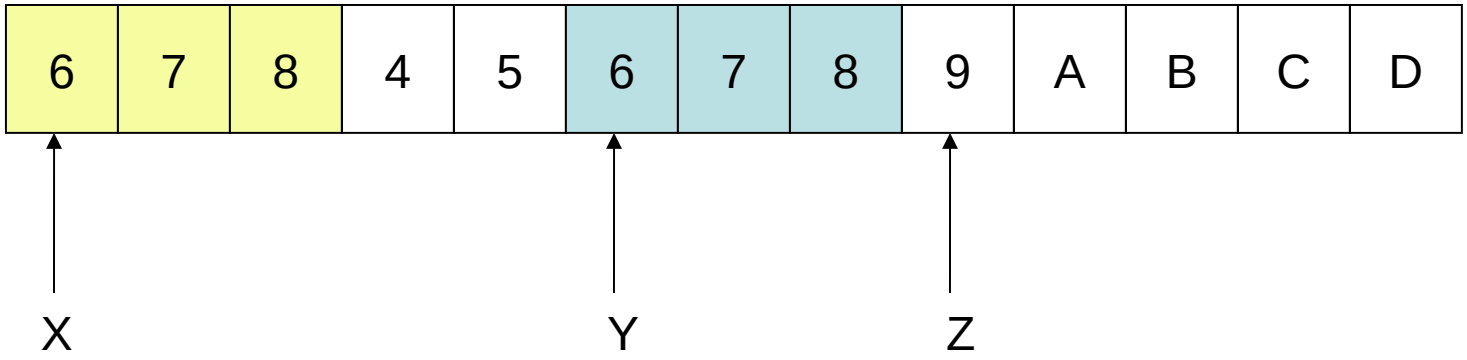
```
X       DC    CL5’12345’
Y       DC    CL3’678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 2

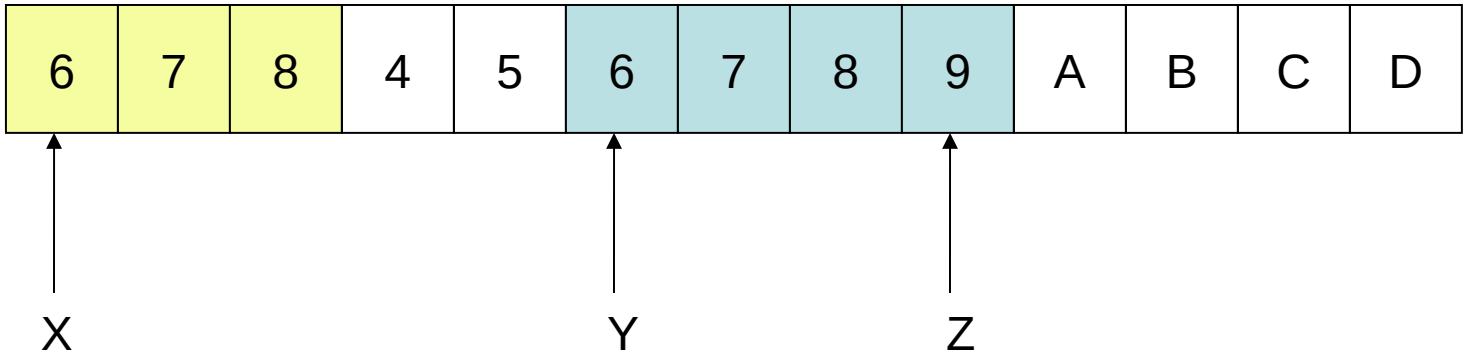
```
X       DC    CL5’12345’
Y       DC    CL3’678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 2

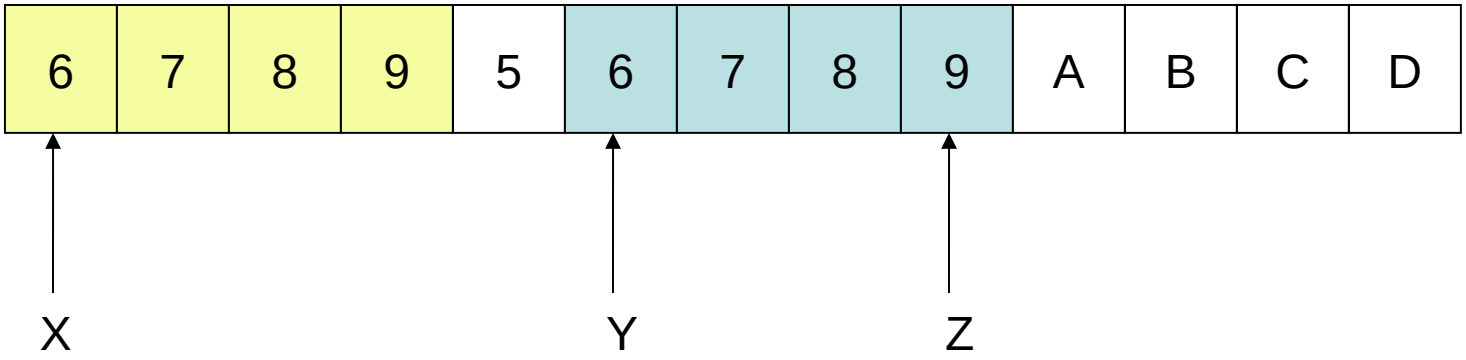
```
X       DC    CL5’12345’
Y       DC    CL3’678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 2

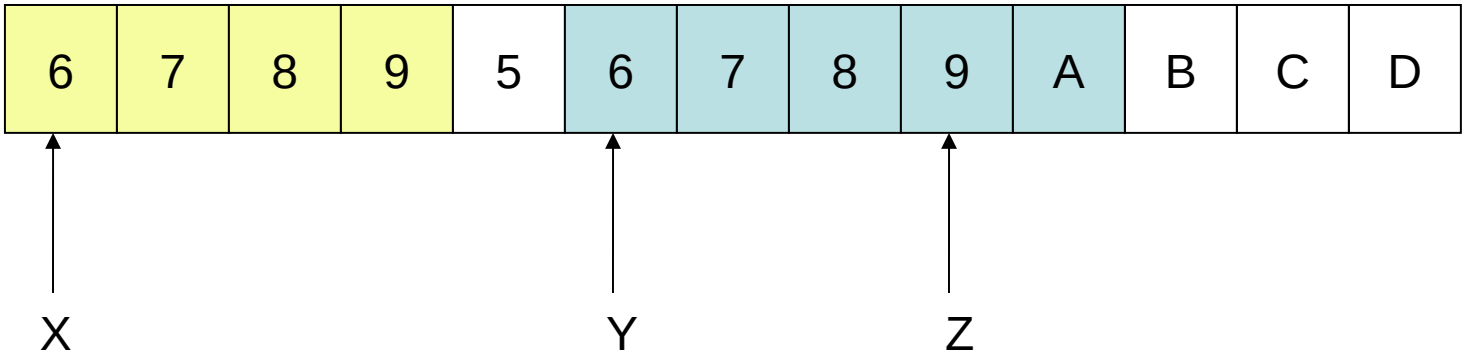
```
X       DC    CL5’12345’
Y       DC    CL3’678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 2

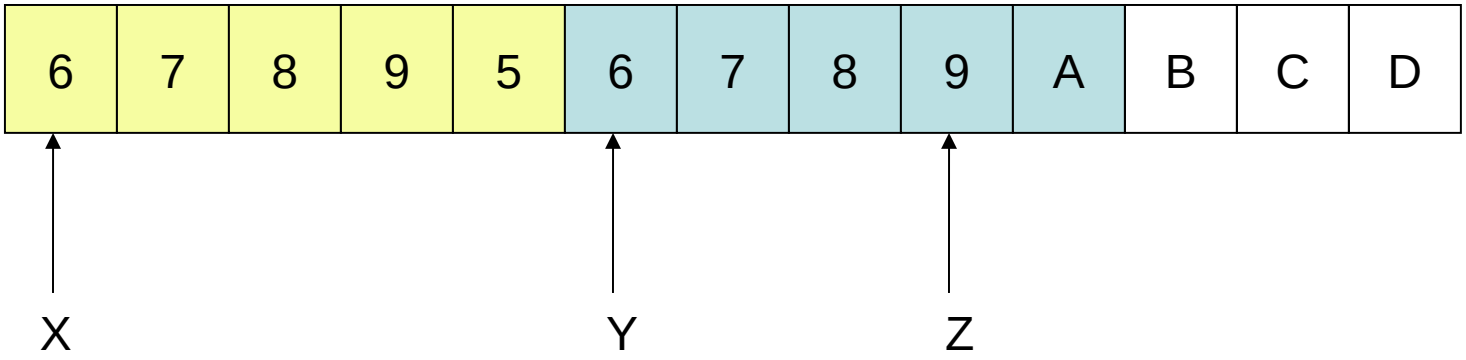
```
X       DC    CL5’12345’
Y       DC    CL3’678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 2

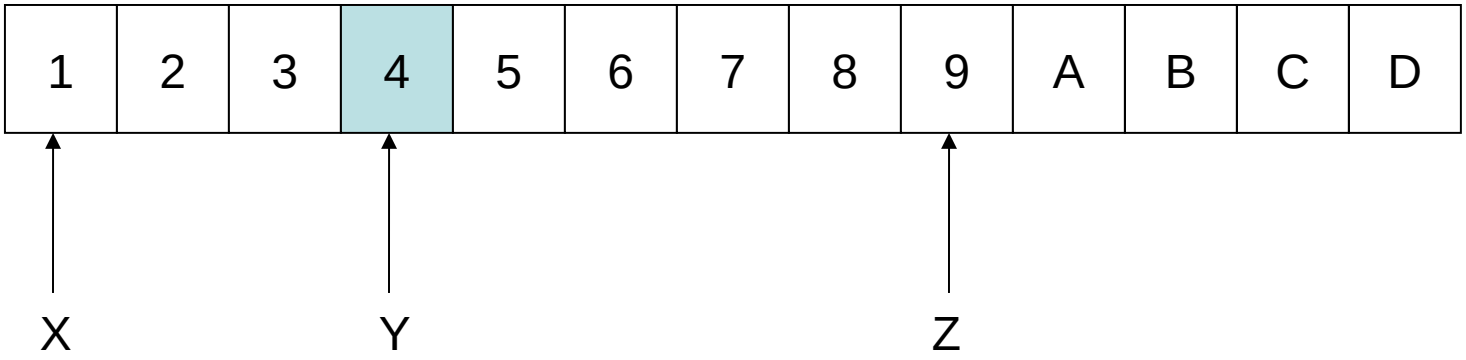
```
X       DC    CL5’12345’
Y       DC    CL3’678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 3

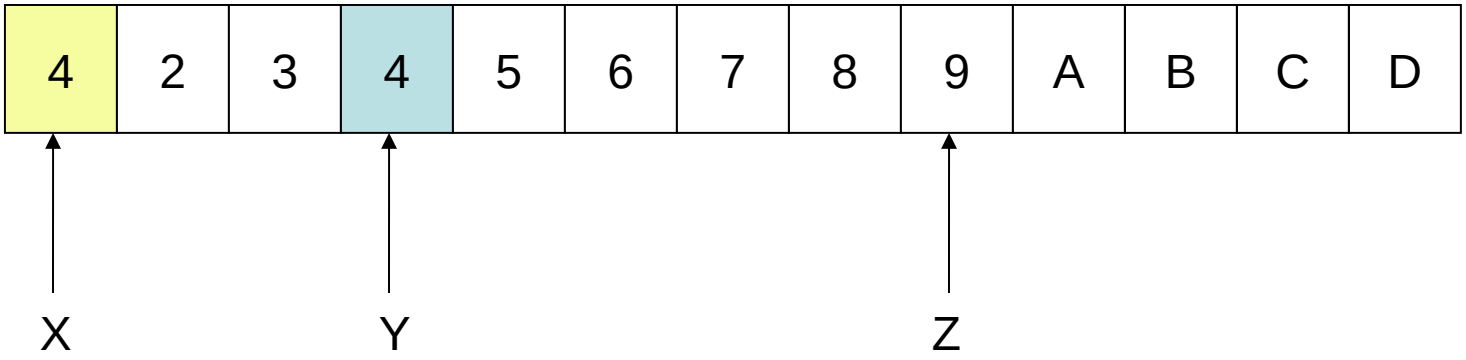
```
X       DC    CL3’123’
Y       DC    CL5’45678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 3

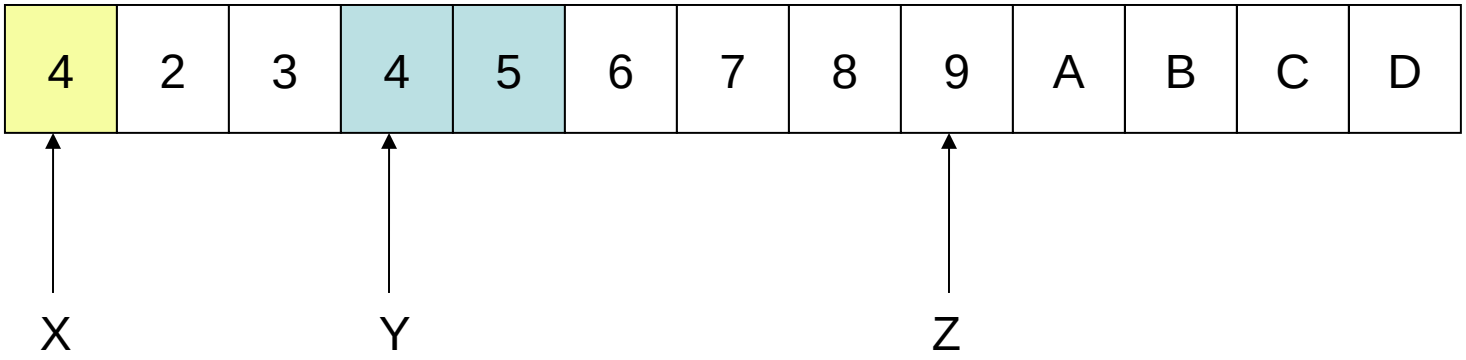
```
X       DC    CL3’123’
Y       DC    CL5’45678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 3

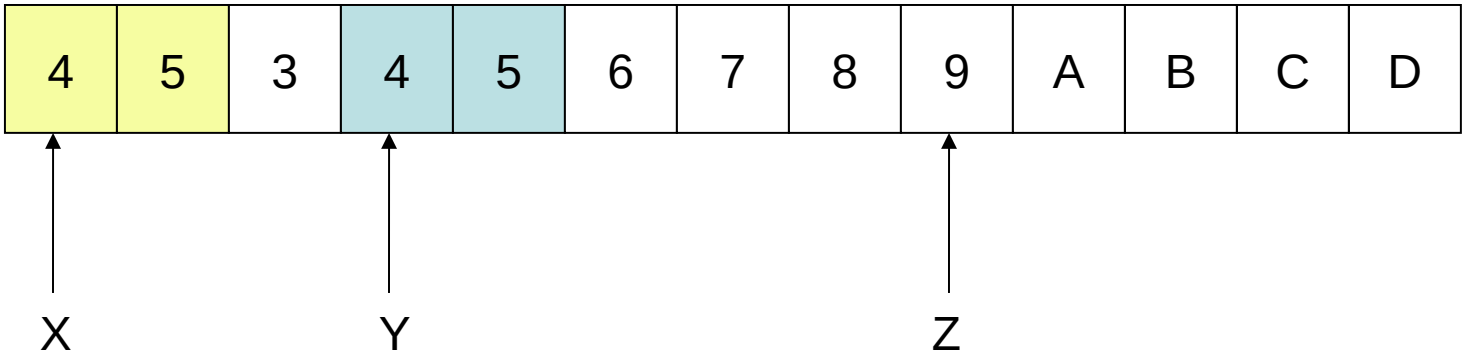
```
X       DC    CL3’123’
Y       DC    CL5’45678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 3

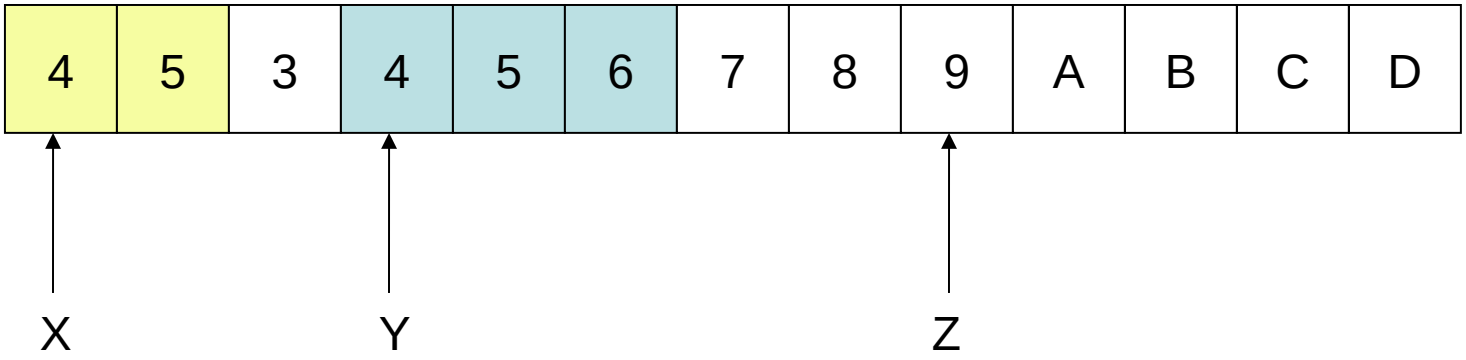
```
X       DC    CL3’123’
Y       DC    CL5’45678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 3

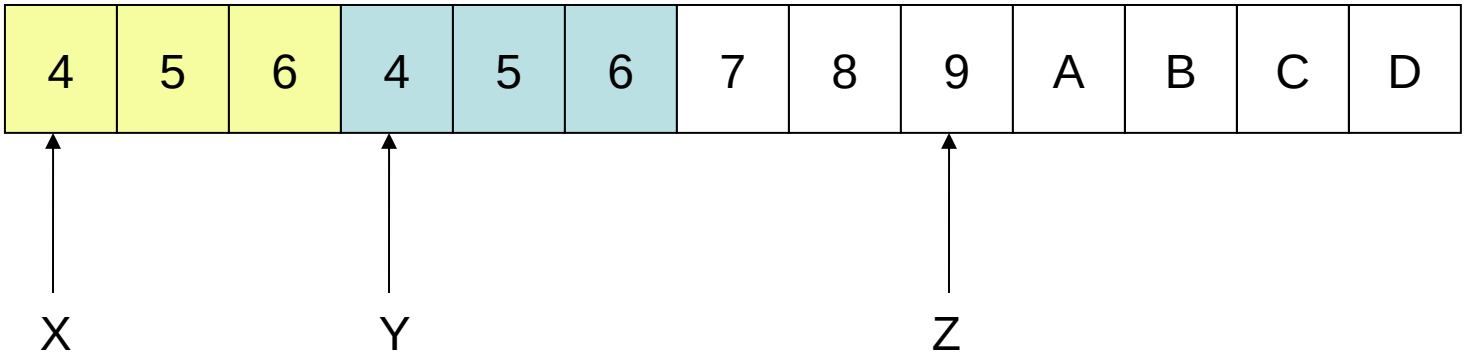
```
X       DC    CL3’123’
Y       DC    CL5’45678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 3

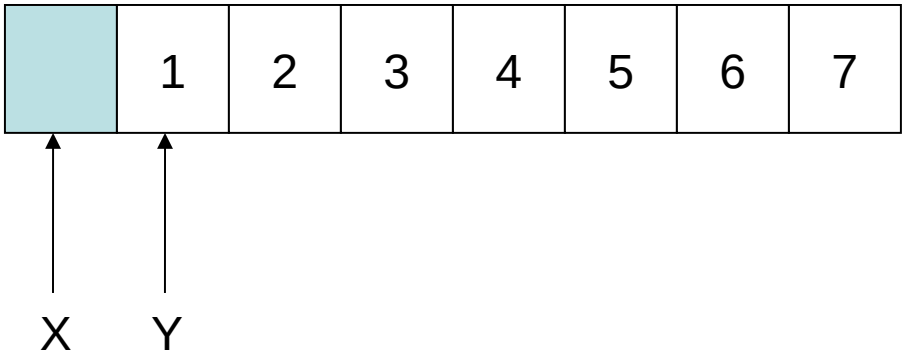
```
X       DC    CL3’123’
Y       DC    CL5’45678’
Z       DC    CL5’9ABCD’
```




```
MVC     X,Y
```

MVC EXAMPLE 4

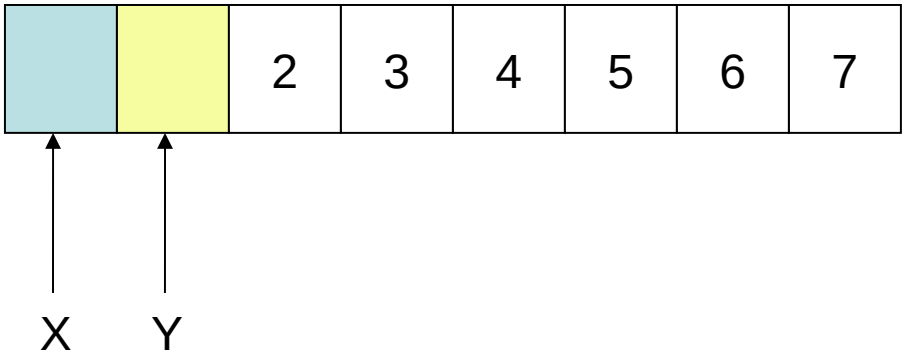
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

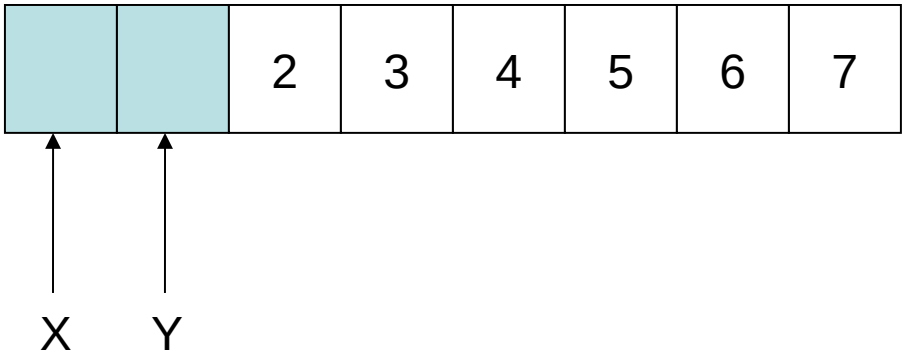
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

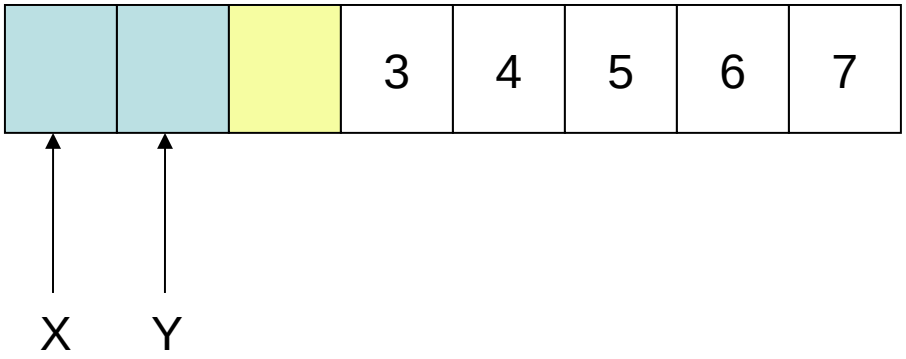
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

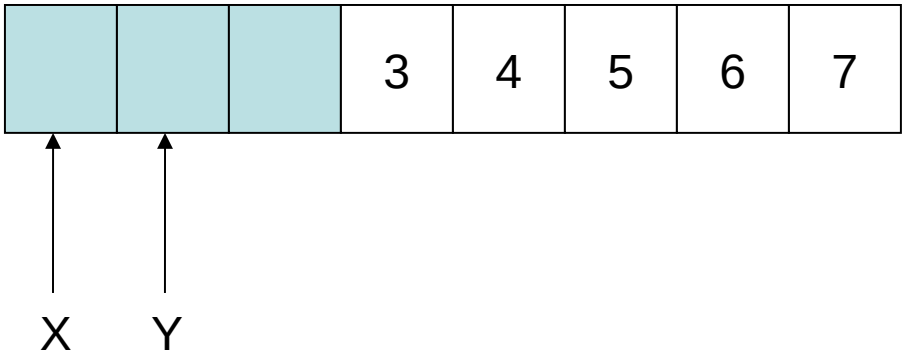
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

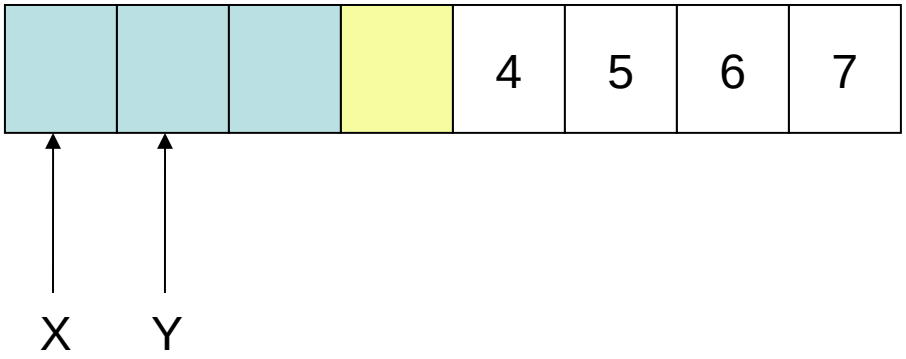
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

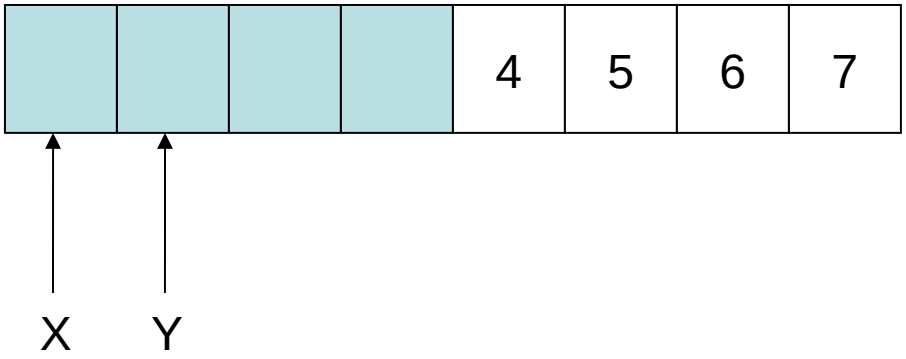
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

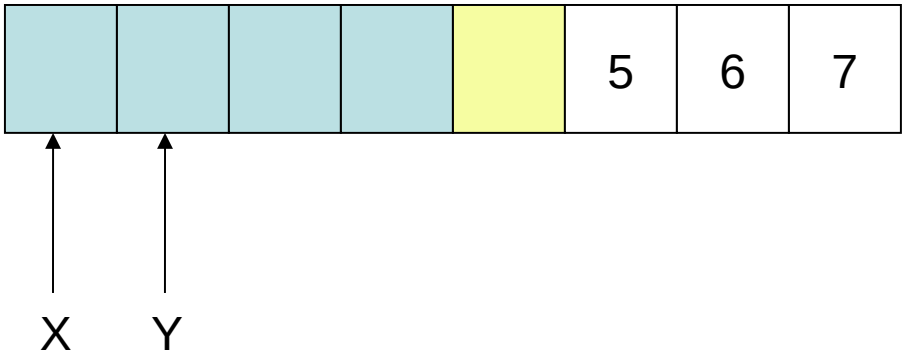
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

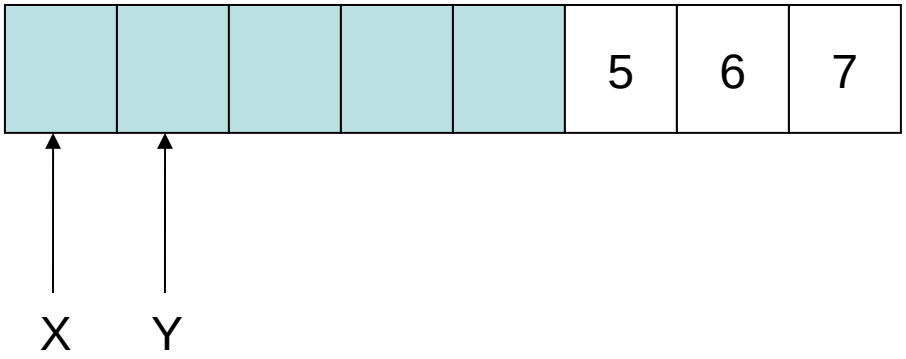
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

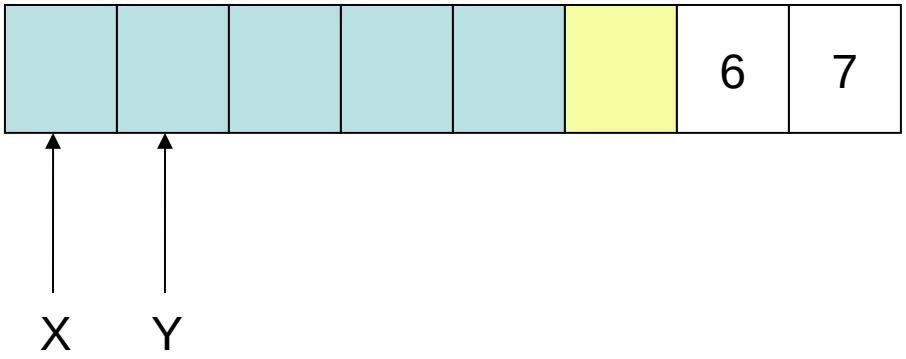
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

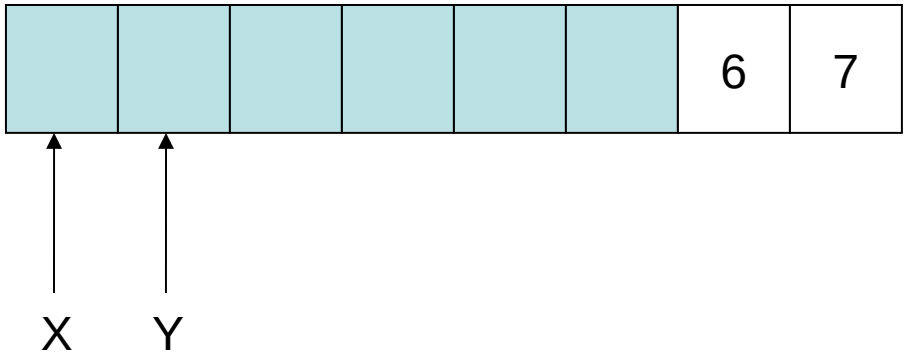
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

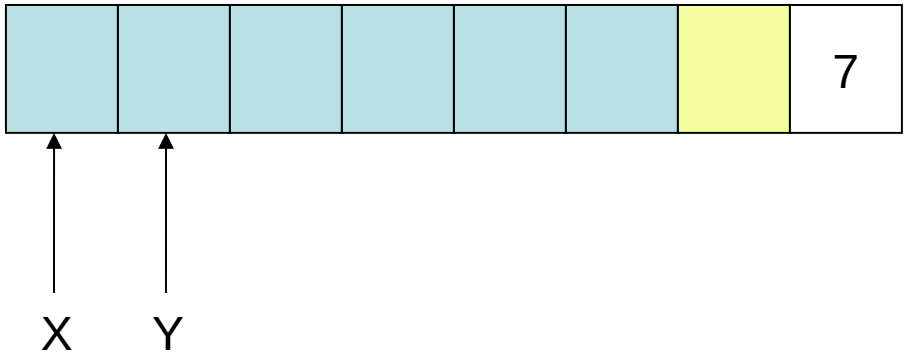
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

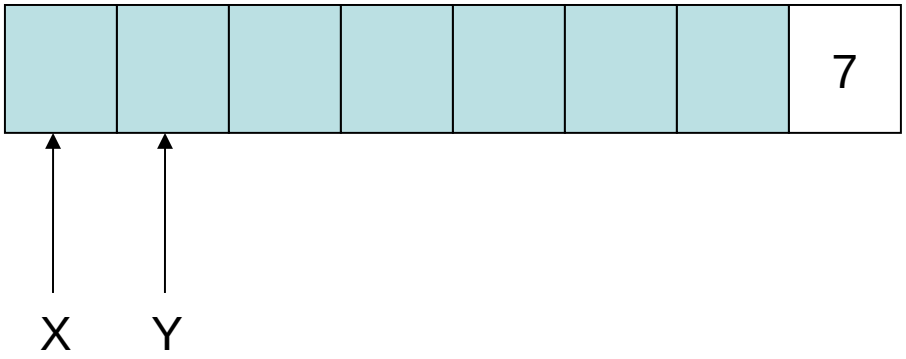
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

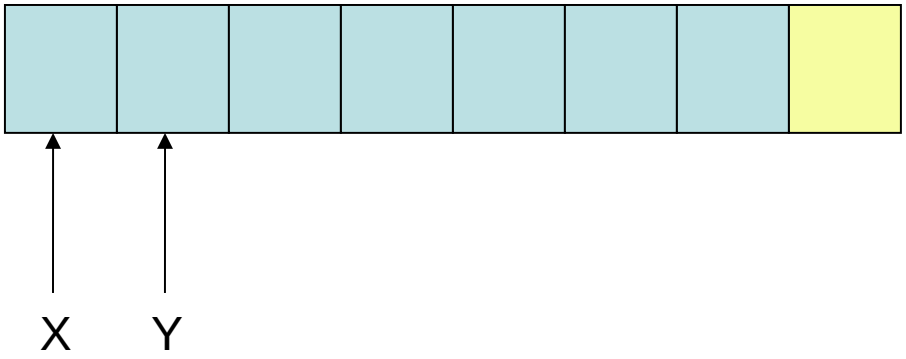
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

MVC EXAMPLE 4

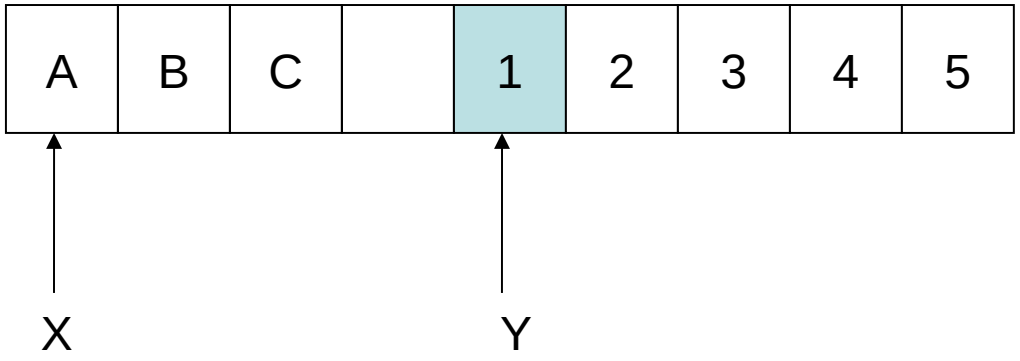
```
X       DC    C' '
Y       DC    CL7'1234567'
```




```
MVC     Y,X
```

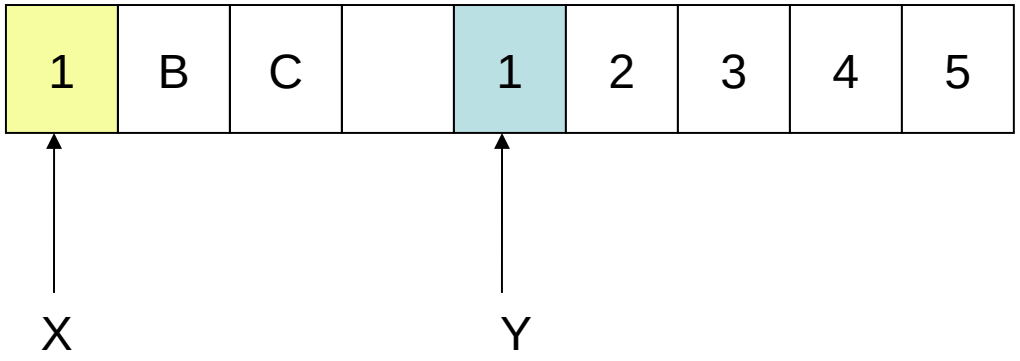
Character Instructions - MVC

```
X    DC      CL4'ABC'
Y    DC      CL5'12345'
     MVC     X,Y
```



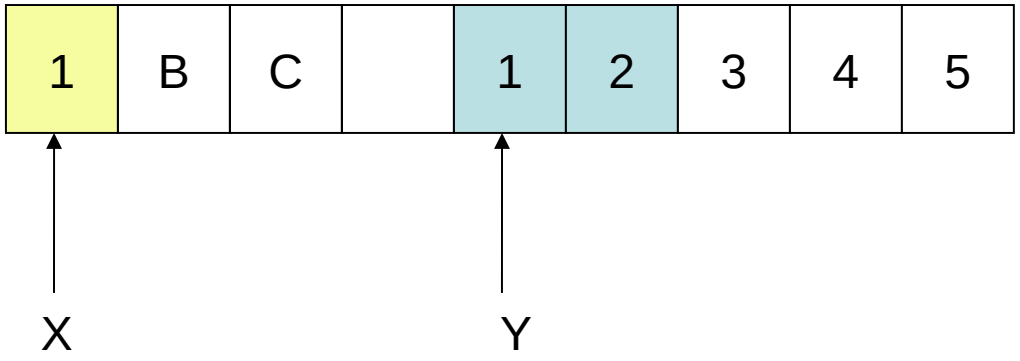
Character Instructions - MVC

```
X    DC      CL4'ABC'
Y    DC      CL5'12345'
     MVC     X,Y
```



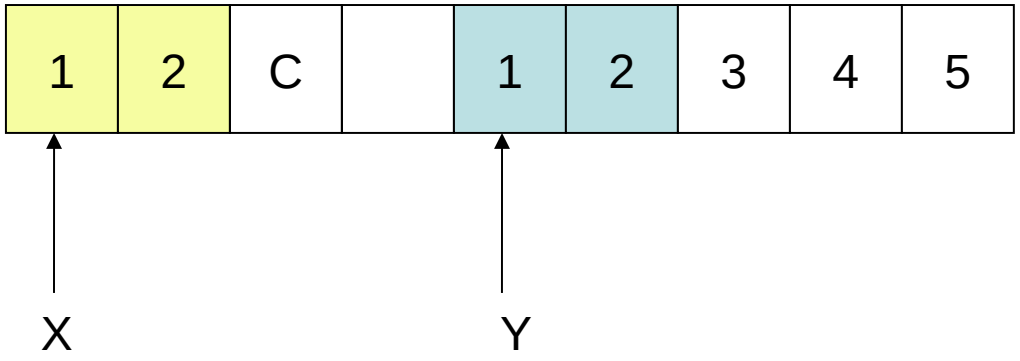
Character Instructions - MVC

```
X    DC      CL4'ABC'
Y    DC      CL5'12345'
     MVC     X,Y
```



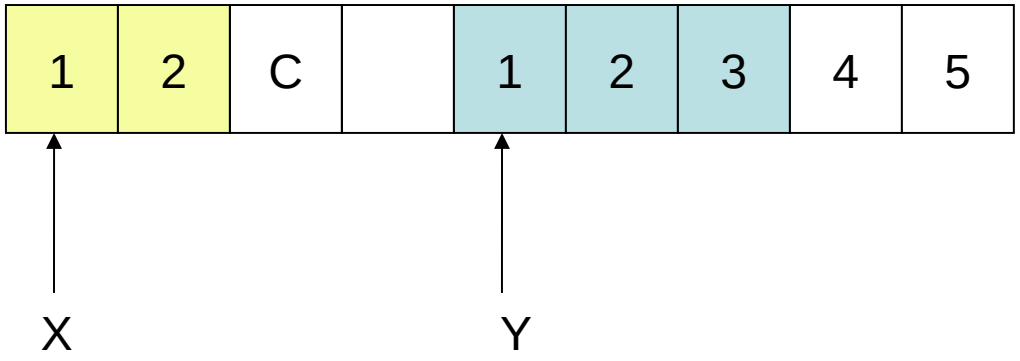
Character Instructions - MVC

```
X     DC        CL4'ABC'
Y     DC        CL5'12345'
      MVC       X,Y
```



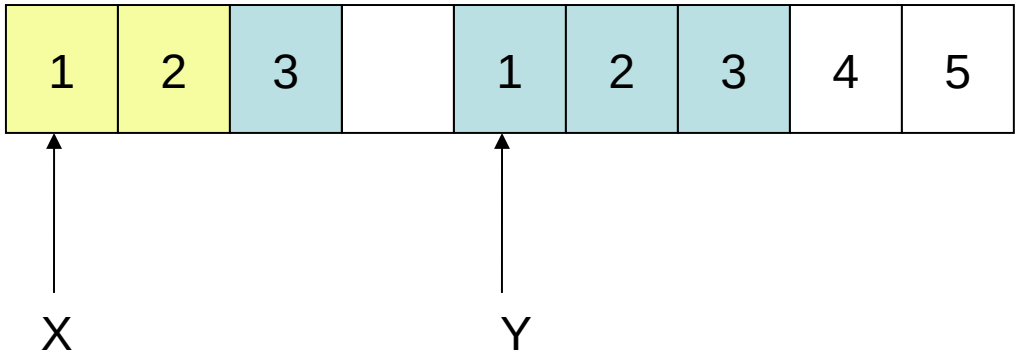
Character Instructions - MVC

```
X     DC        CL4'ABC'
Y     DC        CL5'12345'
      MVC       X,Y
```



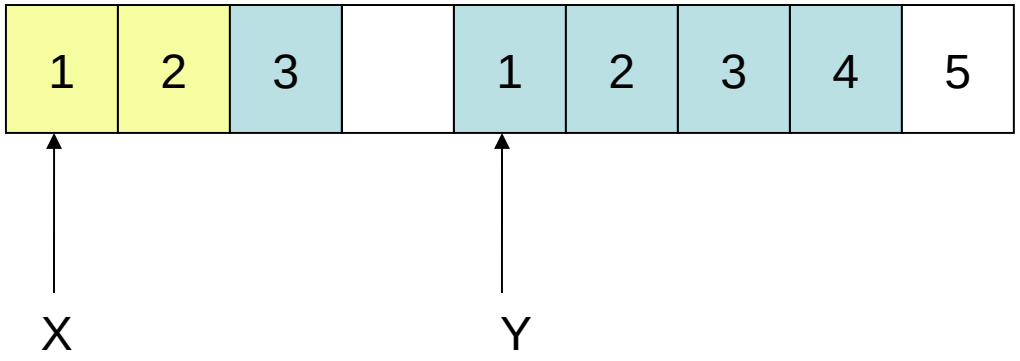
Character Instructions - MVC

```
X     DC        CL4'ABC'
Y     DC        CL5'12345'
      MVC       X,Y
```



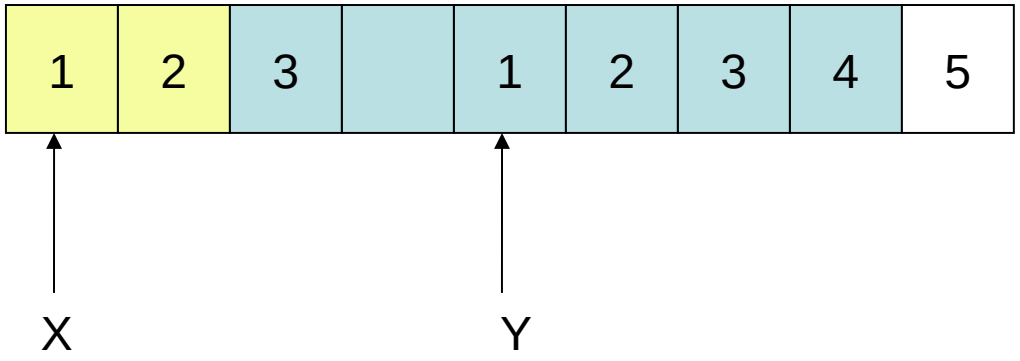
Character Instructions - MVC

```
X     DC        CL4'ABC'
Y     DC        CL5'12345'
      MVC       X,Y
```



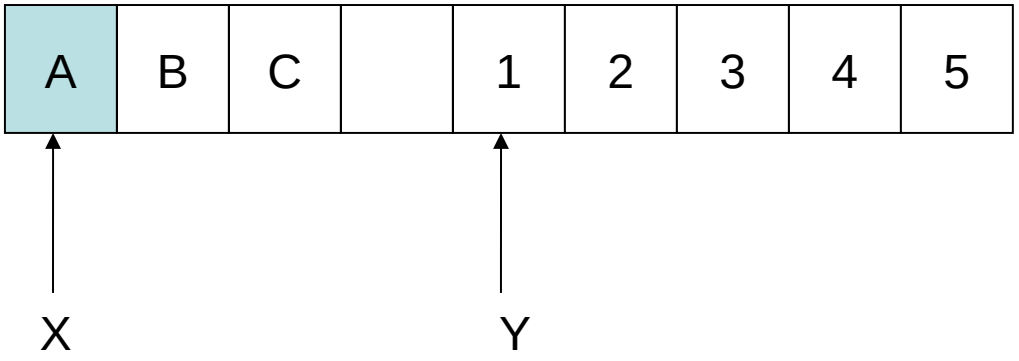
Character Instructions - MVC

```
X     DC        CL4'ABC'
Y     DC        CL5'12345'
      MVC       X,Y
```



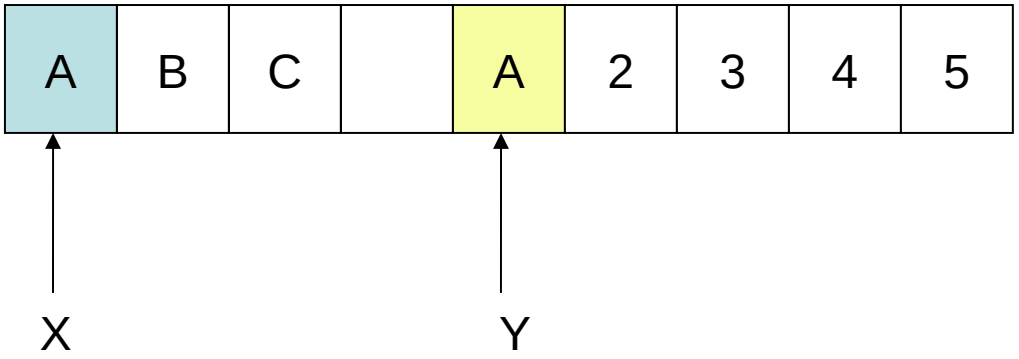
Character Instructions - MVC

```
X    DC      CL4'ABC'
Y    DC      CL5'12345'
     MVC     Y,X
```



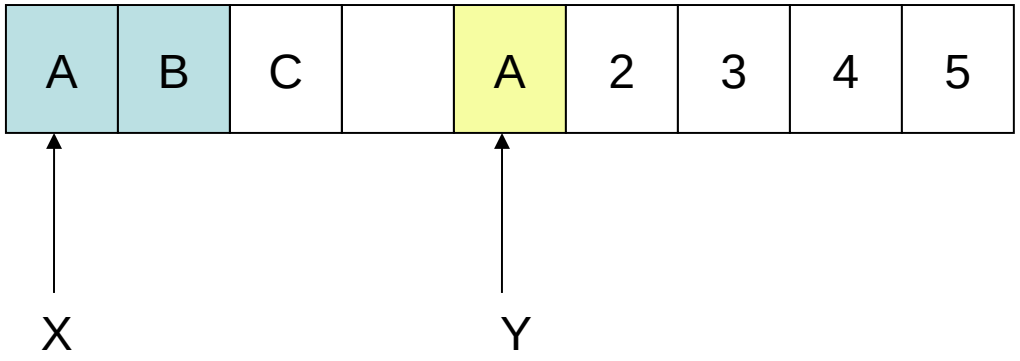
Character Instructions - MVC

```
X    DC      CL4'ABC'
Y    DC      CL5'12345'
     MVC     Y,X
```



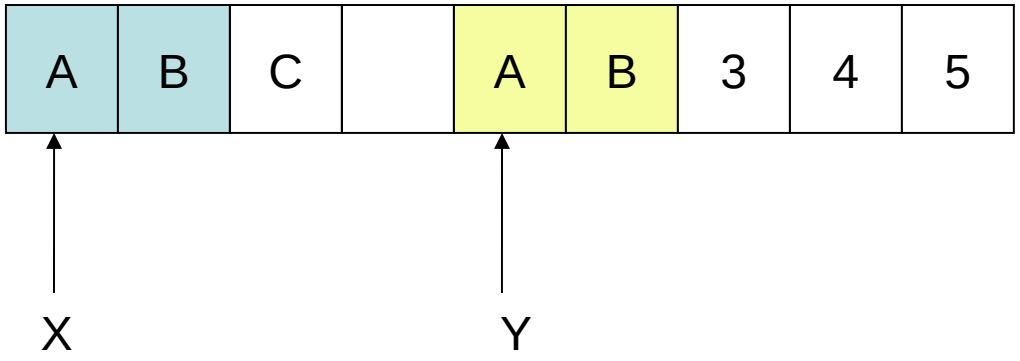
Character Instructions - MVC

```
X     DC        CL4'ABC'
Y     DC        CL5'12345'
      MVC       Y,X
```



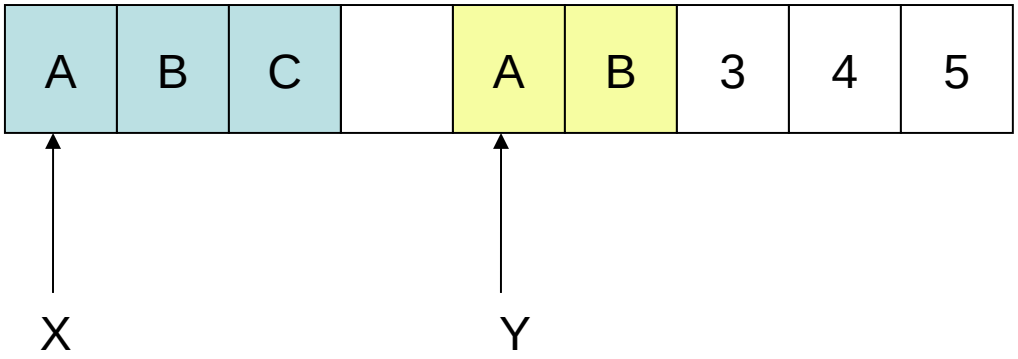
Character Instructions - MVC

```
X    DC      CL4'ABC'
Y    DC      CL5'12345'
     MVC     Y,X
```



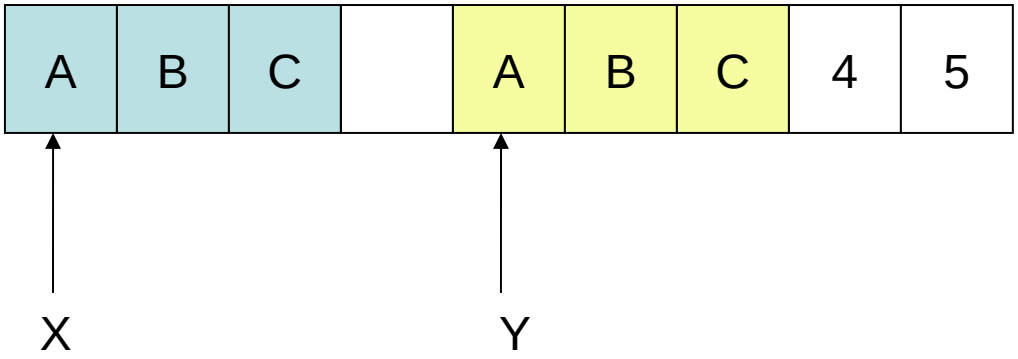
Character Instructions - MVC

```
X     DC        CL4'ABC'
Y     DC        CL5'12345'
      MVC       Y,X
```



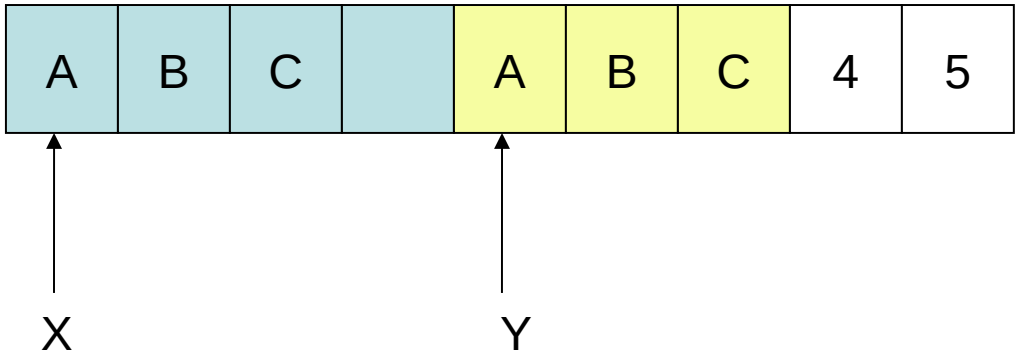
Character Instructions - MVC

```
X    DC      CL4'ABC'
Y    DC      CL5'12345'
     MVC     Y,X
```



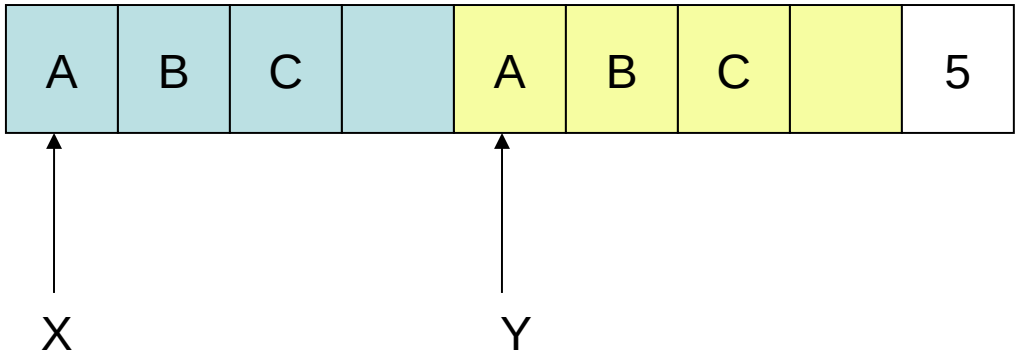
Character Instructions - MVC

```
X    DC      CL4'ABC'
Y    DC      CL5'12345'
     MVC     Y,X
```



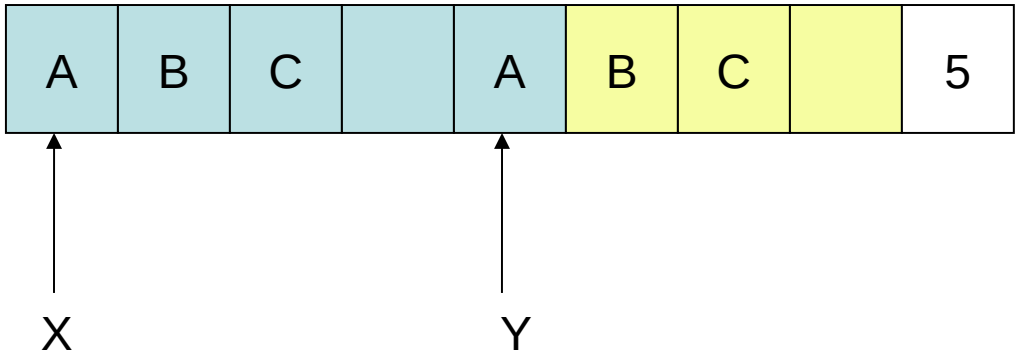
Character Instructions - MVC

```
X    DC      CL4'ABC'
Y    DC      CL5'12345'
     MVC     Y,X
```



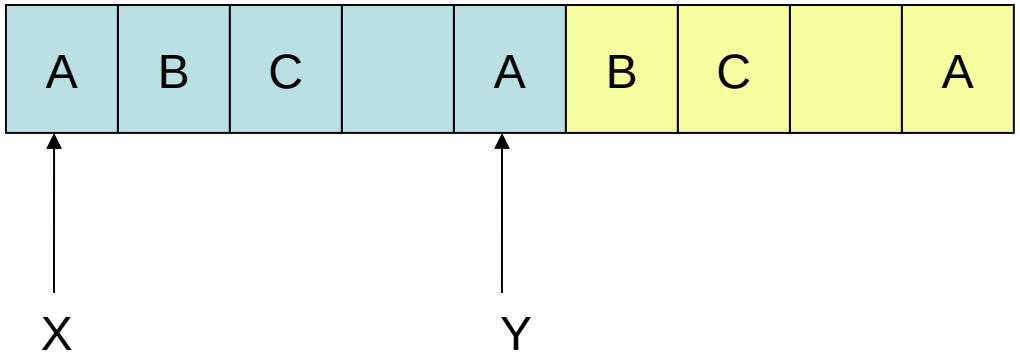
Character Instructions - MVC

```
X    DC      CL4'ABC'
Y    DC      CL5'12345'
     MVC     Y,X
```



Character Instructions - MVC

```
X    DC      CL4'ABC'
Y    DC      CL5'12345'
     MVC     Y,X
```



Character Instructions - MVC

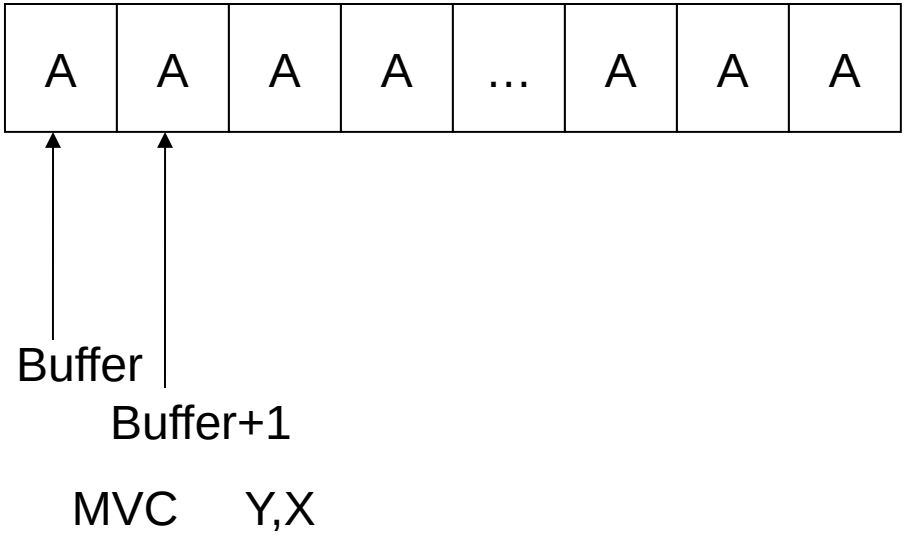
```
X    DC       Cl7’ABC’
Y    DC       CL5’12345’

     MVC      X(3),Y  explicit length
     MVC      X,X+1   relative address
     MVC      X,=C’XYZ’ Careful! why?
```

Blanking Out a Line

```
BLANK     DC          C' '
BUFFER    DS          CL133

MVC    BUFFER,BLANK

MVC    BUFFER,=CL133' '
```

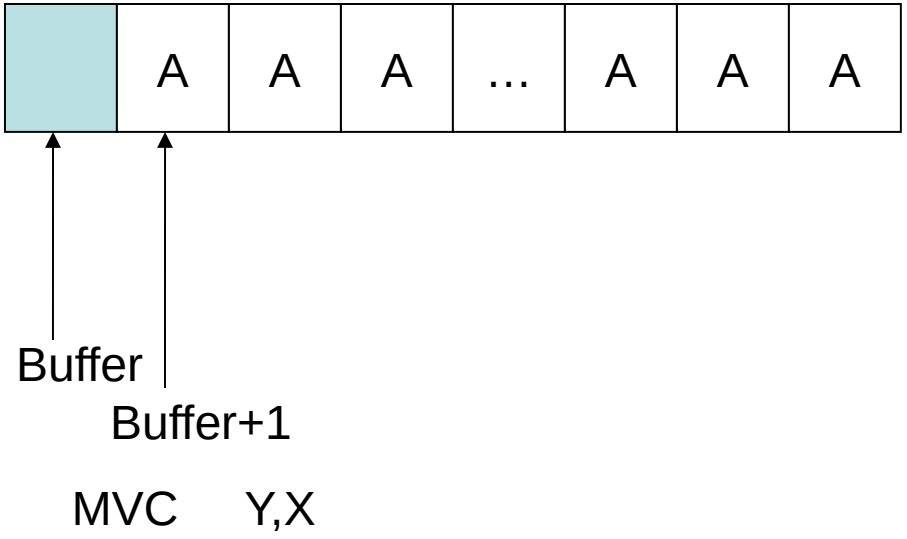
Blanking Out a Line

```
BUFFER    DS        0CL133
          DS        133CL1'A'

MVI       BUFFER,C' '
MVC       BUFFER+1(L'BUFFER-1),BUFFER
```



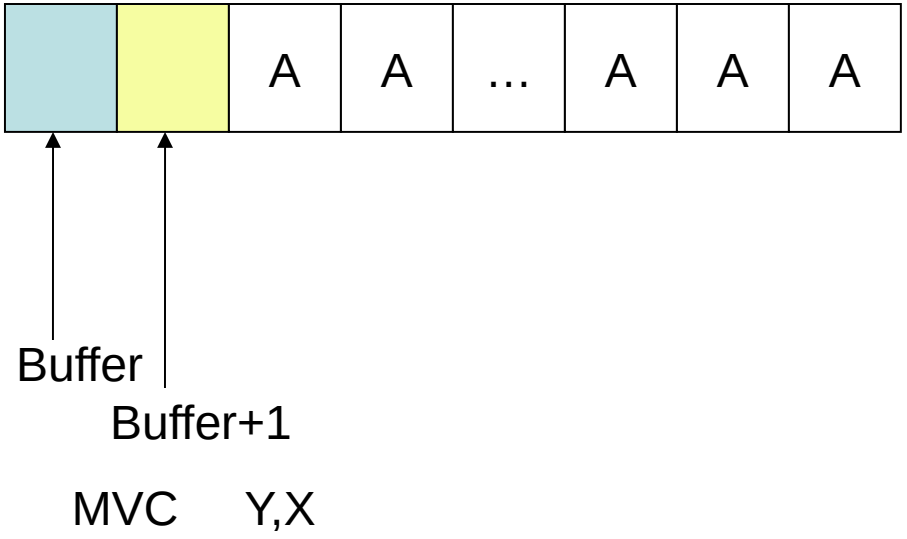
Blanking Out a Line

```
BUFFER    DS        0CL133
          DS        133CL1'A'

MVI       BUFFER,C' '
MVC       BUFFER+1(L'BUFFER-1),BUFFER
```



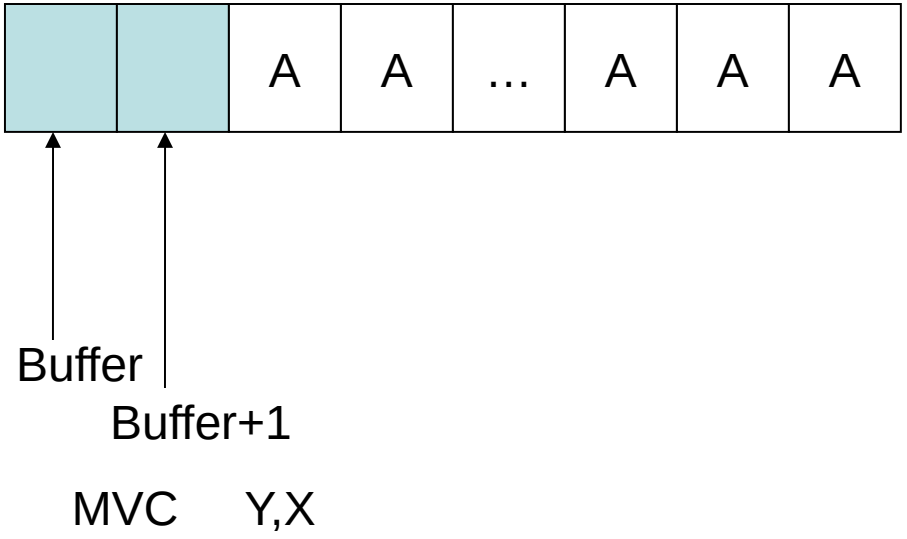
Blanking Out a Line

```
BUFFER    DS        0CL133
          DS        133CL1'A'

MVI       BUFFER,C' '
MVC       BUFFER+1(L'BUFFER-1),BUFFER
```



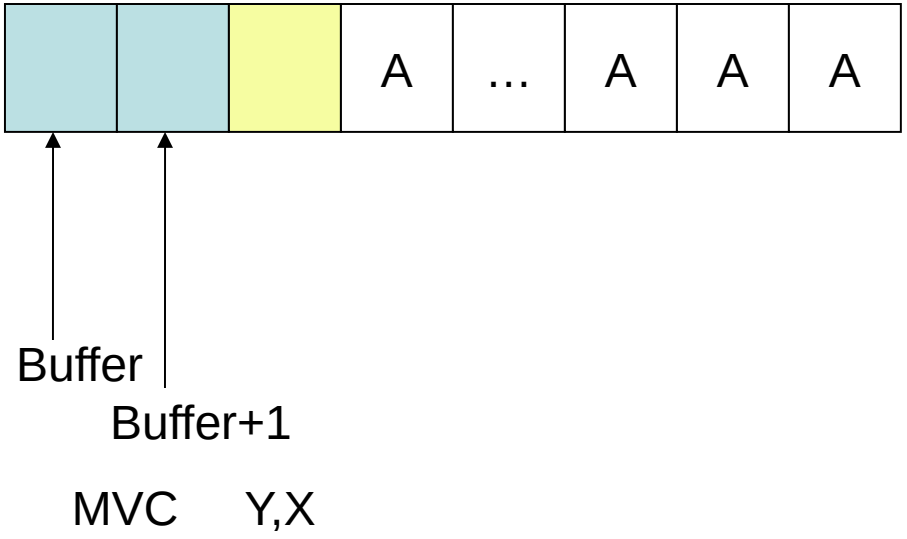
Blanking Out a Line

```
BUFFER    DS        0CL133
          DS        133CL1'A'

MVI       BUFFER,C' '
MVC       BUFFER+1(L'BUFFER-1),BUFFER
```



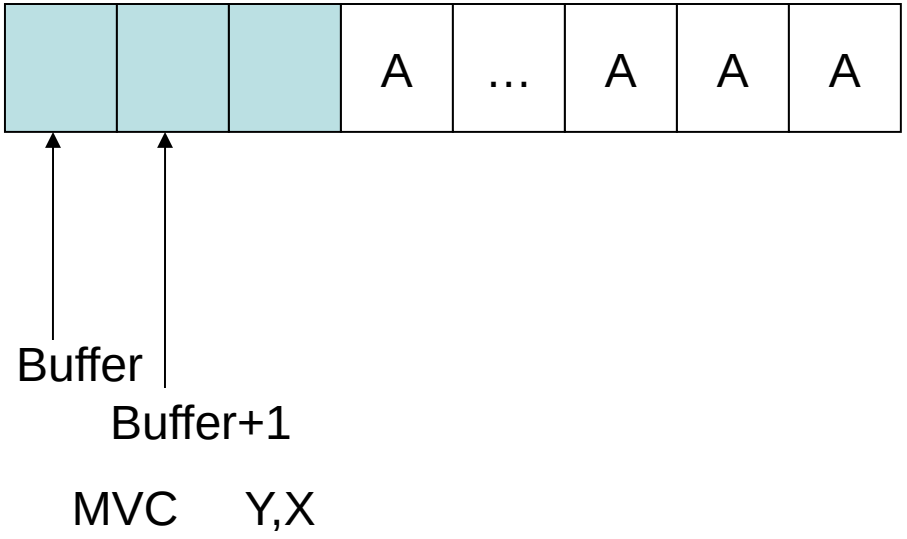
Blanking Out a Line

```
BUFFER    DS        0CL133
          DS        133CL1'A'

MVI       BUFFER,C' '
MVC       BUFFER+1(L'BUFFER-1),BUFFER
```



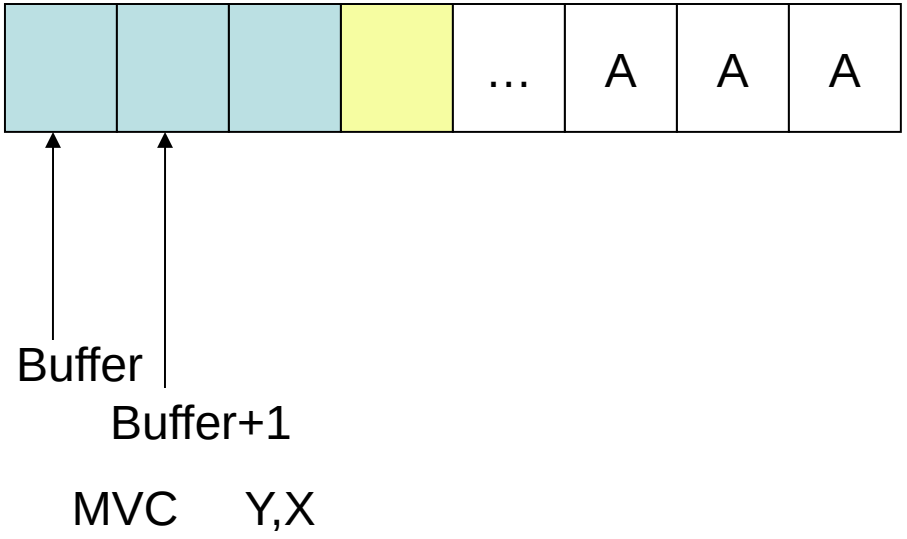
Blanking Out a Line

```
BUFFER    DS        0CL133
          DS        133CL1'A'

MVI       BUFFER,C' '
MVC       BUFFER+1(L'BUFFER-1),BUFFER
```



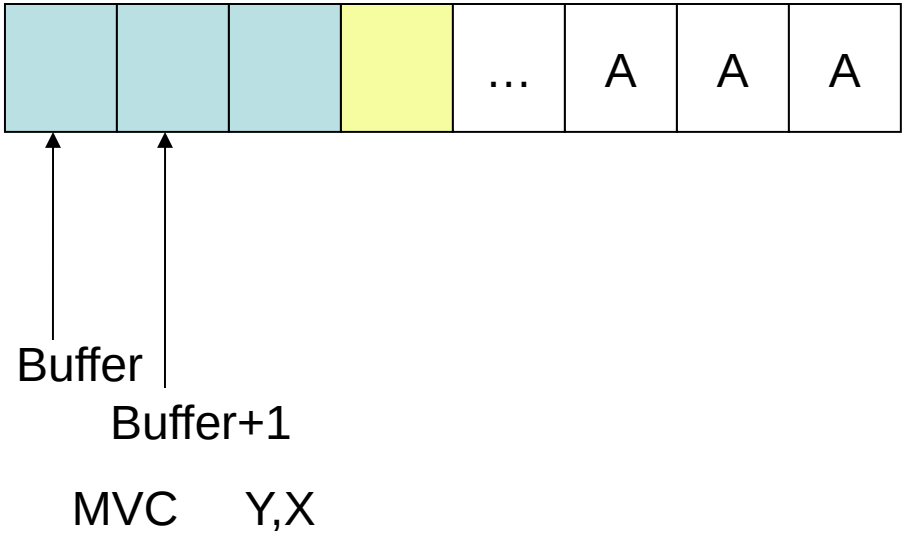
Blanking Out a Line

```
BUFFER    DS        0CL133
          DS        133CL1'A'

MVI       BUFFER,C' '
MVC       BUFFER+1(L'BUFFER-1),BUFFER
```



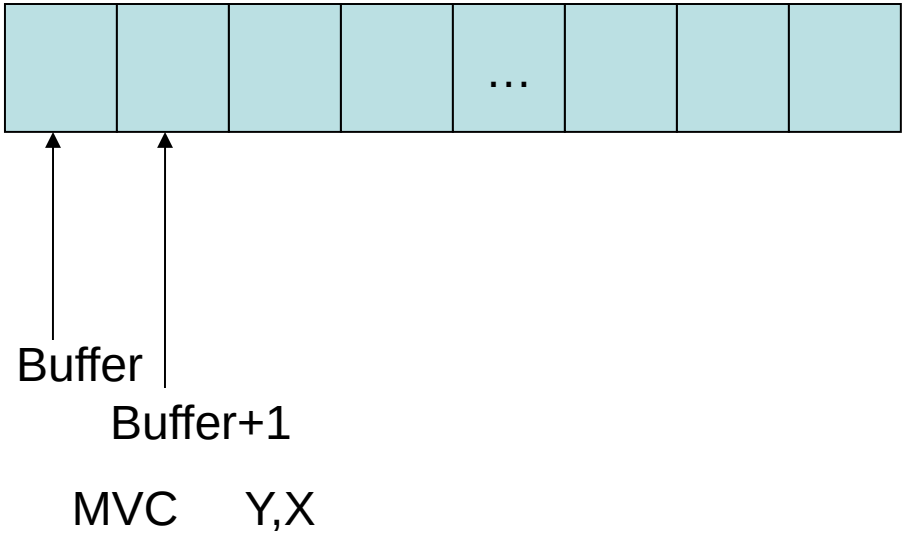
Blanking Out a Line

```
BUFFER    DS        0CL133
          DS        133CL1'A'

MVI       BUFFER,C' '
MVC       BUFFER+1(L'BUFFER-1),BUFFER
```



Blanking Out a Line

```
BUFFER    DS        0CL133
          DS        133CL1'A'

MVI       BUFFER,C' '
MVC       BUFFER+1(L'BUFFER-1),BUFFER
```



Character Instructions - CLC

- Compare Logical Characters
- SS_1
- Sets the condition code to indicate how Op1 compares to Op2
- Length – associated with operand 1 only!
- Default Length is length of operand 1
- Bytes compared using EBCDIC encoding
- Bytes compared left to right
- Max 256 bytes are compared
- Test with BE, BL, BH, BNE, BNL, BNH

Character Instructions - CLC

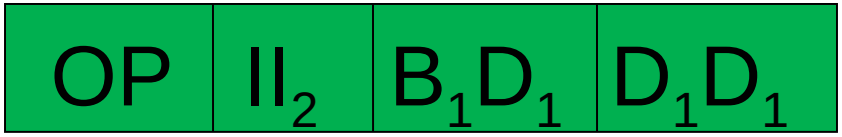
```
X       DC    C'ABC'
Y       DC    C'1234567'
Z       DC    C'ABCD'
        CLC   X,Y       cc = low
        BE    THERE     no branch
        ------------------------
        CLC   Y,X       cc = high
        BNE   THERE     branch taken
        ------------------------
        CLC   X(2),Z    cc = equal
        BNE   THERE     no branch taken
        ------------------------
        CLC   Z,X       cc = low
        BNH   THERE     branch taken
```

Character Instructions - CLI

- Compare Logical Immediate
- SI
- Constant resides inside the instruction
- Sets the condition code to indicate how Op1 compares to Op2
- Implicit length of 1!
- Byte compared using EBCDIC encoding
- Test with BE, BL, BH, BNE, BNL, BNH

Object Instruction Formats

- Storage Immediate (SI)
  General Object Format:



- OP – Operation code
- II_2 – Immediate Constant – Operand 2
- $B_1D_1D_1D_1$ – Base/Disp of Operand 1

Character Instructions - CLI

```
X     DC      Cl3'ABC'
Y     DC      CL7'1234567
Z     DC      C'ABCD'
      CLI     X,C'A'     cc = equal
      CLI     Y,C'A'     cc = high
      CLI     X,X'40'    cc = high
      CLI     Z,C'B'     cc = low
      CLI     Y,B'11110001' cc = equal
      CLI     Y,C'1'     cc = equal
      CLI     Y,X'F1'    cc = equal
      BL      THERE
```

Character Instructions - MVI

- Move Immediate
- SI
- Constant resides inside the instruction
- Moves one byte to operand one
- Implicit length of 1!

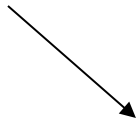
Character Instructions - MVI

```
X       DC      CL5'ABC'
Y       DC      CL7'1234567'
Z       DC      C'ABCD'
        MVI     X,C'F'
        MVI     Y,C'A'
        MVI     X,X'40'
        MVI     Z,64
        MVI     Z,B'11110001'
```

Defining an Output File

```
*****************************************************************
*
*      OUTPUT FILE - DATA CONTROL BLOCK
*
*****************************************************************
FILEOUT  DCB   DSORG=PS,                                        X
         MACRF=(PM),                                            X
         DDNAME=FILEOUT,                                        X
         RECFM=FB,                                              X
         LRECL=80
```

Column 72



DCB – Data Control Block – storage used to describe a file
DSORG=PS – Data Set Organization – Physical Sequential
MACRF=(PM) – Output File – Move the record from my program to an output buffer
DDNAME=FILEOUT – Data Definition Name – JCL File Name
RECFM=FB – Record Format – Fixed Blocked
LRECL=80 – Logical Record Length in bytes

Defining an Input File

```
*********************************************************************
*
*      INPUT FILE - DATA CONTROL BLOCK
*
*********************************************************************
FILEIN   DCB   DSORG=PS,                                           X
         MACRF=(GM),                                               X
         DDNAME=FILEIN,                                            X
         RECFM=FB,                                                 X
         EODAD=FINAL                                               X
         LRECL=80
```

MACRF=(GM) Macro Format – Input file - Get a record,move it to my program

RECFM=FB – Record Format – Fixed blocked

EODAD – End Of Data Address – where do we branch when we try to read a record from an empty file?

LRECL – the number of bytes in a logical record

DCB (Data Control Block) Parms

- MACRF=(GM)
  - Macro format – controls the type of macros that are used for file
  - GM – input file, move mode
  - GL – input file, locate mode
  - PM – output file, move mode
  - PL – output file, locate mode
  - We will start with GM and PM since these are the easiest, but GL and PL are the most efficient for files with large record sizes and many records
- DDNAME= name
  - The JCL DD name

DCB (Data Control Block) Parms

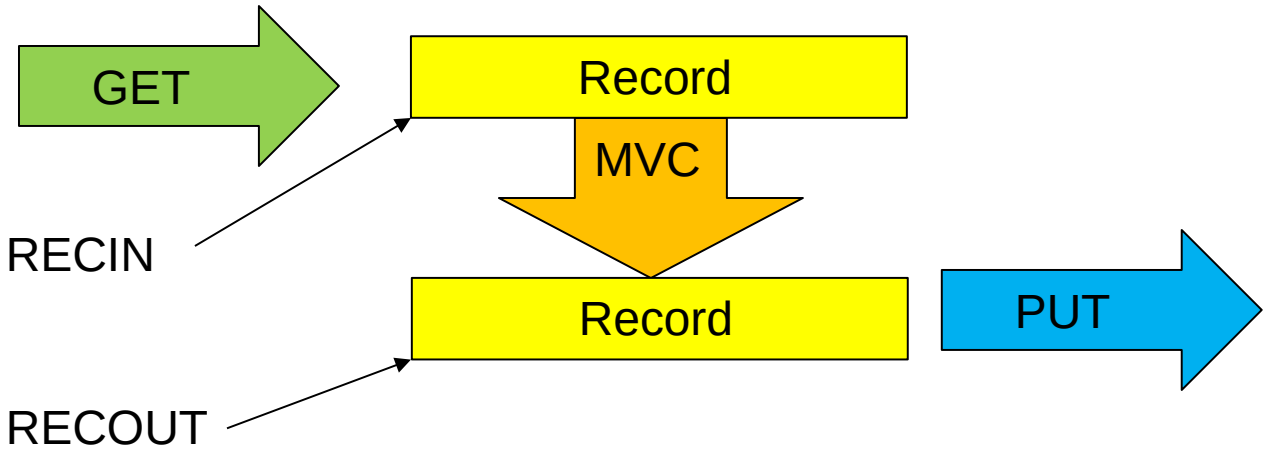
- DSORG=PS
  - A Physical sequential file organization
- RECFM=(FB)
  - The record format
  - FB – fixed blocked
  - FBA – fixed blocked with ANSI carriage control characters
  - FBM – fixed blocked with IBM carriage control characterss
  - VB – variable blocked
  - F – fixed unblocked
  - V – variable unblocked
- EODAD=label
  - End of Data Address. Branch here when EOF.
- LRECL=length
  - Logical Record Length. Take default on the blocksize – let the OS choose.

Opening and Closing Files

```
OPEN  (FILEIN,(INPUT))          THE INPUT FILE
OPEN  (FILEOUT,(OUTPUT))        MY REPORT

... (process the files)

CLOSE FILEIN
CLOSE FILEOUT
```

Reading and Writing Records

```
*   MOVE MODE IO
         GET   FILEIN,RECIN
         MVC   MESSAGEO,MESSAGEI
         PUT   FILEOUT,RECOUT
```

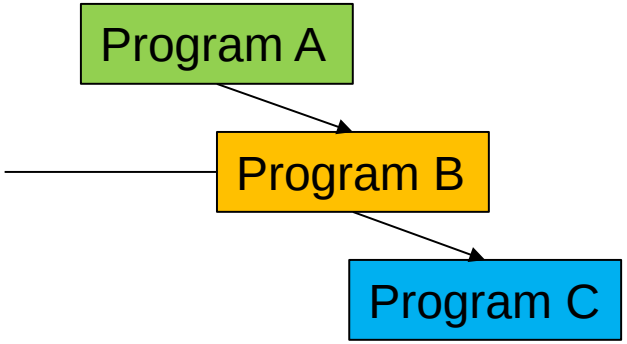


Standard Entry and Exit

```
SKELAD4  CSECT
* STANDARD ENTRY
         STM   R14,R12,12(R13)      STANDARD ENTRY
         BASR  R12,R0
         USING *,R12
         ST    R13,SAVE+4
         LA    R13,SAVE
         L     R13,SAVE+4
         …
* STANDARD EXIT
         LM    R14,R12,12(R13)
         LA    R15,0
         BR    R14
```

Save Area (18F)

Used by Pl/I
Program A's Save Area Address
Program C's Save Area Address
Register 14
Register 15
Register 0
Register 1
Register 2
Register 3
Register 4
Register 5
Register 6
Register 7
Register 8
Register 9
Register 10
Register 11
Register 12



Exercise #1

- Read a sequential dataset in which each record contains a single date in the following format:

  YYYYMMDD (columns 1-8)
- You can assume valid data.
- Print Three records in the output file for each input record
- Reformat the date data in the following format:

  Year: 2004

  Month: January

  Day: 23
- The numeric month should be converted to English. Don't try to be fancy here, just use brute force for the conversions – we don't know much assembler at this point – a string of comparisons will work.

Exercise #1

Here is some sample data:

20161128
20160115
20160215
20160315
20160415
20160515
20160615
20160715
20160815
20160915
20161015
20162115
20162215
20160102
20160130
19441207

Object Instruction Formats

- Storage to Storage (SS2)

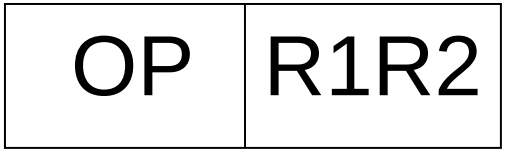
  General Object Format:

OP	L1L2	B1D1	D1D1	B2D2	D2D2

- OP – Operation code
- L1 – Length of Operand One - Max value 15
- L2 – Length of Operand Two - Max value 15
- B1D1D1D1 – Base/Disp of Operand 1
- B2D2D2D2 – Base/Disp of Operand 2

Object Instruction Formats

- Register to Register (RR)
  General Object Format:



- OP – Operation code
- R1 – Operand 1 register
- R2 – Operand 2 register

Object Instruction Formats

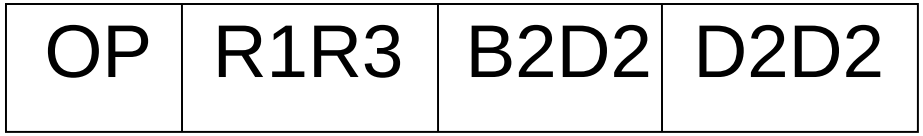
- Register to Indexed Storage (RX)
  General Object Format:

OP	R1X2	B2D2	D2D2

- OP – Operation code
- R1 – Operand 1 Register
- X2 – Operand 2 Register
- B2D2D2D2 – Base/Disp of Operand 2

Object Instruction Formats

- Register to Storage (RS)
  General Object Format:



- OP – Operation code
- R1 – Operand 1 Register
- R3 – Operand 3 Register or Mask
- B2D2D2D2 – Base/Disp of Operand 2

Explicit Instruction Format

- Storage to Storage (SS2)

  Explicit Instruction format

  OP D1(L1,B1),D2(L2,B2)
- OP – Operation code
- D1, D2 – Displacement of operand 1 and 2
- B1, B2 – Base regs of operand 1 and 2
- L1 – Length of operand 1 (Max 256)
- L2 – Length of operand 2 (Max 256)

Explicit Instruction Format

- Register to Register (RR)

  Explicit Instruction format

  OP R1, R2
- OP – Operation code
- R1 – Register for operand 1
- R2 – Register for operand 2

Explicit Instruction Format

- Storage Immediate (SI)

  Explicit Instruction format

  OP D1(B1),I2
- OP – Operation code
- D1 – Displacement for operand 1
- B1 – Base Register for operand 1
- I2 – Immediate Constant for operand 2

Explicit Instruction Format

- Register to Indexed Storage (RX)

  Explicit Instruction format

  OP R1,D2(X2,B2)
- OP – Operation code
- R1 – Register for operand 1
- D2 – Displacement for operand 2
- B2 – Base Register for operand 2
- X2 – Index Register for operand 2

Explicit Instruction Format

- Register to Storage (RS)

  Explicit Instruction format

  OP R1,R3,D2(B2)
- OP – Operation code
- R1 – Register for operand 1
- R3 – Register or mask for operand 3
- D2 – Displacement for operand 2
- B2 – Base Register for operand 2

Reading Explicit Instructions

```
MVC     3(4,5),6(7)          SS1
AP      3(4,5),6(7,8)        SS2
MVI     4(8),X'40'           SI
LR      5,12                 RR
STM     14,12,12(13)         RS
L       6,9(5,8)             RX
```

Reading Mixed Instructions

```
X           DS      CL8
Y           DS      CL20

            MVC     X(3),6(7)               SS1
            MVC     X(L'Y),Y                SS1
            AP      X(3),Y(8)               SS2
            MVI     X,X'40'                 SI
            LR      R5,R12                  RR
            STM     R14,R12,SAVE            RS
            L       R6,X(8)                 RX
```

Packed Decimal Data

- Two decimal digits per byte
- Sign occurs in the numeric portion of the rightmost byte
- Valid signs: A,C,F are +
  B, D are -
- Preferred signs: C (+), D (-)
- XPK DC PL3’54’
- Generates: 00054C

Example Packed Data Fields

```
APK      DC    P'12345'          12345C
BPK      DC    PL2'12345'          345C
CPK      DC    PL4'123.45'     0012345C
DPK      DC    P'-345'             345D
EPK      DC    P'0'                 0C
FPK      DC    P'-1'                1D
```

Zoned Decimal Data

- One decimal digit per byte
- Sign occurs in the zone portion of the rightmost byte
- Valid signs: A,C,F are +
  B, D are -
- Preferred signs: C (+), D (-)
- XPK DC ZL5’-354’
  Generates: F0F0F3F5D4
- Some character fields qualify as zoned:
- X DC C’12345’
  Generates: F1F2F3F4F5

Example Zoned Data Fields

```
AZD      DC   Z'12345'     F1F2F3F4C5
BZD      DC   ZL2'-12345'        F4D5
CZD      DC   Z'-12345'    F1F2F3F4D5
DZD      DC   ZL4'-345'      F0F3F4D5
EZD      DC   Z'0'                C0
FZD      DC   Z'-1'               D1
GC       DC   C'12345'     F1F2F3F4F5
```

Object Instruction Formats

- Storage to Storage (SS2)

  General Object Format:

OP	L_1L_2	B_1D_1	D_1D_1	B_2D_2	D_2D_2

- OP – Operation code
- L_1 – Length of Operand One - Max value 15
- L_2 – Length of Operand Two - Max value 15
- $B_1D_1D_1D_1$ – Base/Disp of Operand 1
- $B_2D_2D_2D_2$ – Base/Disp of Operand 2

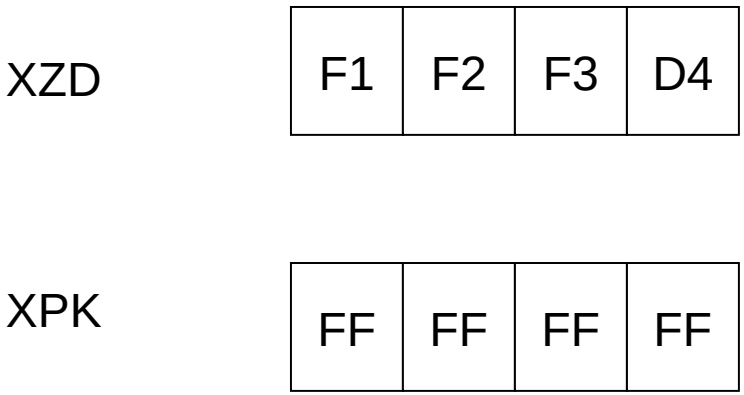
Packed Instructions - Pack

- SS2
- The PACK instruction converts from zoned decimal to packed decimal

```
XZD     DC      Z'-1234'    F1F2F3D4
XPK     DS      PL4         0001234D
        …
        PACK    XPK,XZD
```

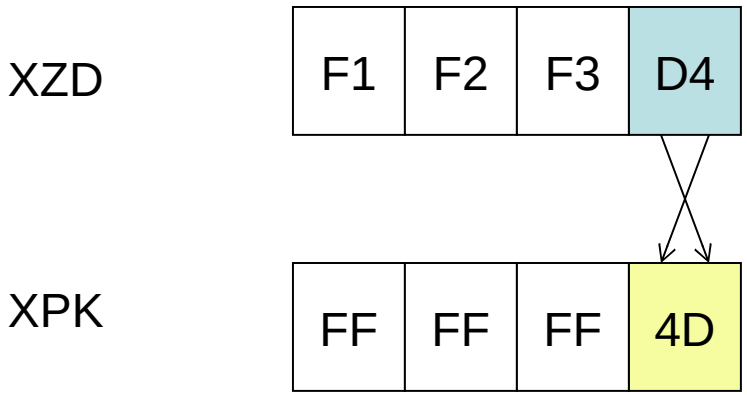
Packed Instructions - Pack

```
XZD    DC     Z'-1234'   F1F2F3D4
XPK    DS     PL4        0001234D
       …
       PACK   XPK,XZD
```



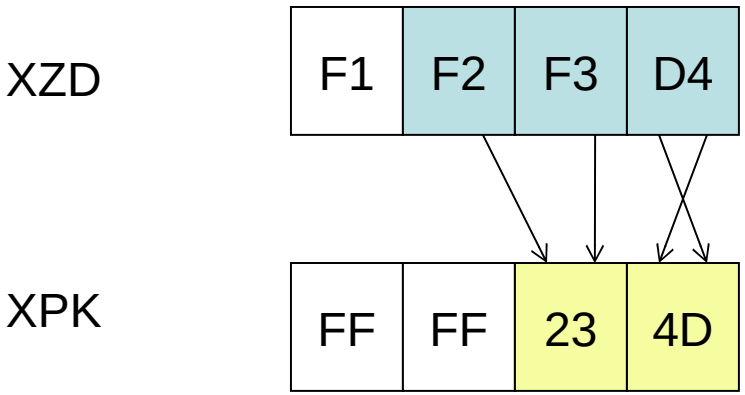
Packed Instructions - Pack

```
XZD    DC     Z'-1234'    F1F2F3D4
XPK    DS     PL4         0001234D
       …
       PACK   XPK,XZD
```



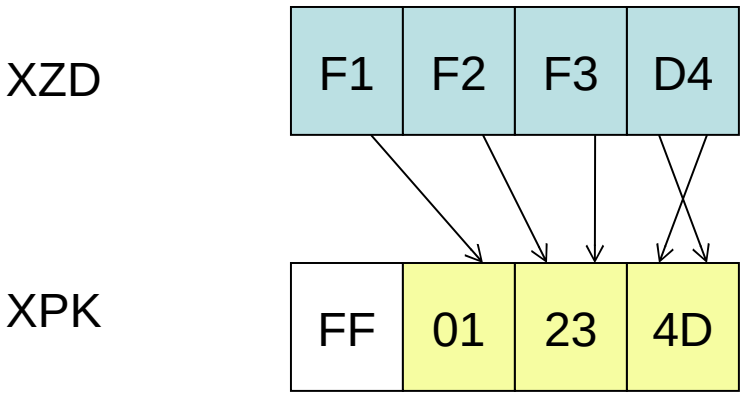
Packed Instructions - Pack

```
XZD     DC      Z'-1234'    F1F2F3D4
XPK     DS      PL4         0001234D
        …
        PACK    XPK,XZD
```



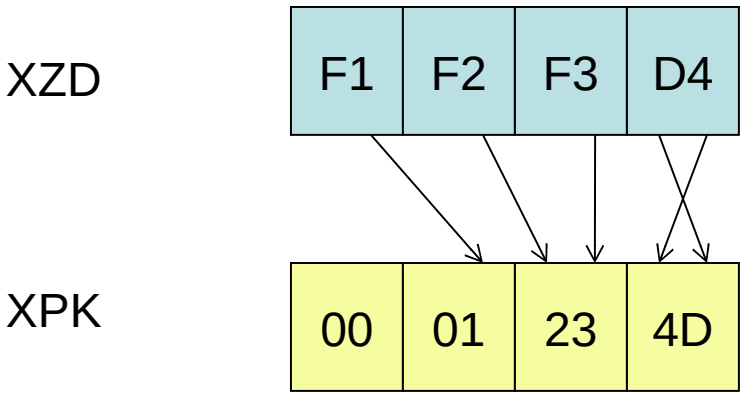
Packed Instructions - Pack

```
XZD     DC      Z'-1234'    F1F2F3D4
XPK     DS      PL4         0001234D
        …
        PACK    XPK,XZD
```



Packed Instructions - Pack

```
XZD    DC     Z'-1234'    F1F2F3D4
XPK    DS     PL4         0001234D
       …
       PACK   XPK,XZD
```



Packed Instructions - Pack

- The receiving field can be too short

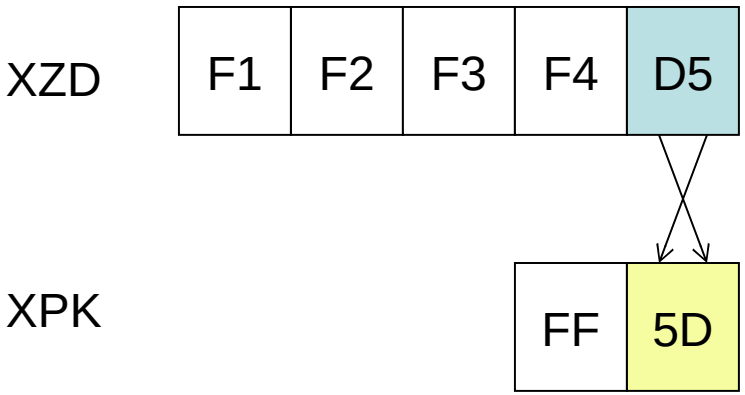
```
XZD DC Z'-12345'      F1F2F3F4D5
XPK DS PL2                  345D
          …
          PACK      XPK,XZD
```

- Any data can be packed – results not always good

Packed Instructions - Pack

- The receiving field can be too short

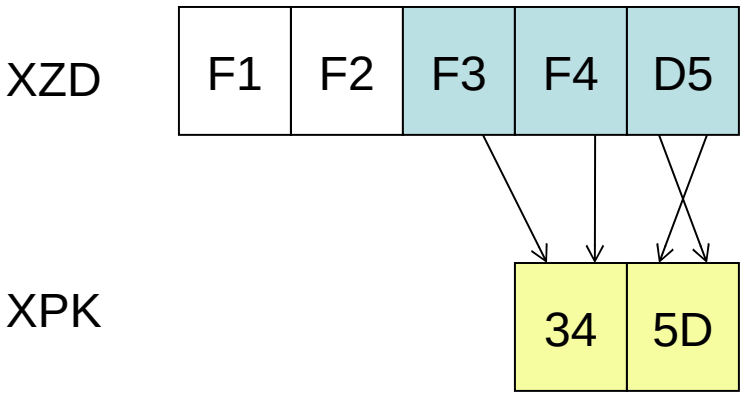
```
XZD DC Z'-12345'     F1F2F3F4D5
XPK DS PL2                345D
```



Packed Instructions - Pack

- The receiving field can be too short

```
XZD DC Z'-12345'      F1F2F3F4D5
XPK DS PL2                  345D
```



Packed Decimal Arithmetic Issues

- Each operand contains at most 16 bytes – 31 digits plus a sign
- Data Exceptions (S0C7) occur when performing arithmetic on data that is not packed
- Loss of precision can occur on ZAP, AP, SP – no notification

Packed Arithmetic - ZAP

- SS_2
- Use ZAP to move packed data (not MVC)

```
XPK       DS    PL4
YPK       DC    P'54'   054C
          ZAP   XPK,YPK
          MVC   XPK,YPK  WHY NOT?
```

- Final result in XPK: 0000054C
- Sets the condition code to Positive, Minus, or equal
- Test with BP, BM, BE, BNP, BNM, BNE

Packed Arithmetic - AP

- SS_2 – Add packed
- Op1 – Target Field (max 16 bytes)
- Op2 – Source Field (max 16 bytes)
- The target contains the arithmetic sum of Op1 and Op2
- Sets the condition code:
  - High – result > 0 Test with BP, BNP
  - Low – result < 0 Test with BM, BNM
  - Equal – result = 0 Test with BZ, BNZ

Packed Arithmetic - SP

- SS2 – Subtract packed
- Op1 – Target Field (max 16 bytes)
- Op2 – Source Field (max 16 bytes)
- The target contains the arithmetic result of subtracting Op2 from Op1
- Sets the condition code:
  - High – result > 0 Test with BP, BNP
  - Low – result < 0 Test with BM, BNM
  - Equal – result = 0 Test with BZ, BNZ

Packed Arithmetic

```
X       DS    PL8
Y       DC    PL4'123'
        ZAP   X,Y
        AP    X,X
        SP    X,=P'1'
        ZAP   X,X
        BP    THERE
```

Packed Arithmetic - MP

- SS2 – Multiply packed
- Op1 – Target Field (max 16 bytes)
- Op2 – Source Field (max 8 bytes)
- The target contains the product of multiplying Op1 by Op2
- Does not set the condition code
- Op1 contains the multiplicand (before) and then the product (after)
- Op2 contains the multiplier
- The number of leading bytes of zeros in op1 before multiplying must be >= no of bytes in op2

Leaving Enough Room

```
WORK    DS    PL4
A       DC    PL3'12345'
B       DS    PL2

        ZAP   WORK,A
        MP    WORK,B    ABENDS!
WORK:  0012345C
```

Packed Arithmetic

```
X       DS    PL8
Y       DC    PL4'20'
Z       DC    PL4'30
        ZAP   X,Y
        MP    X,Z

        ZAP   X,=PL5'123456789'
        MP    X,Y   PROBLEM
```

Packed Arithmetic - DP

- SS2 – Divide packed
- Op1 – Target Field (max 16 bytes) Dividend/Quotient/Remainder
- Op2 – Source Field (max 8 bytes) Divisor
- The target initially contains the dividend
- After dividing, target contains quotient and remainder
- Remainder size = divisor size
- Quotient size = Target field size – Remainder size
- Does not set the condition code
- The number of leading bytes of zeros in op1 before dividing must be >= no of bytes in op2

Packed Arithmetic

```
X     DS    PL8     COMPUTE 125/30
Y     DC    PL5'125'
Z     DC    PL3'30
      ZAP   X,Y
      DP    X,Z
```

X Before divide: 000000000000125C

X After divide: 000000004C00005C

Quotient Remainder

Example Division

Divide X by Y Giving QUOT Remainder REM

```
X           DS     PL5
Y           DS     PL3
WORK        DS     0PL8
QUOT        DS      PL5
REM         DS      PL3
            ZAP     WORK,X
            DP      WORK,Y
```

Packed Arithmetic - CP

- SS2 – Compare packed
- Op1 – Target Field (max 16 bytes)
- Op2 – Source Field (max 16 bytes)
- Sets the condition code to indicate how Op1 compares with Op2
- Numbers are compared arithmetically (not a character compare with EBCDIC)
- Fields can be different sizes

Packed Arithmetic

```
X       DC    PL8'-30'
Y       DC    PL5'125'
Z       DC    PL3'30'
        CP    X,Y      CC = LOW
        CP    Y,X      CC = HIGH
        CP    Y,=P'125'   CC = EQUAL
        CP    Z,=X'030F'  CC = EQUAL
```

CP Example With Extended Mnemonic

```
        ZAP   XPK,=P'0'
THERE   EQU   *
        AP    XPK,=P'1'
        CP    XPK,=P'10'
        BNE   THERE
        …
XPK     DS    PL3
```

CP Example With Branch on Condition

```
         ZAP   XPK,=P'0'
THERE    EQU   *
         AP    XPK,=P'1'
         CP    XPK,=P'10'
         BC    7,THERE
         …
XPK      DS    PL3
```

Some Extended Mnemonics

Each condition is represented with 4 bits

Equal Zero	Low Minus	High Plus	Overflow		Condition	
0	0	1	0	=	2	BH,BP
1	0	0	0	=	8	BE,BZ
0	1	1	1	=	7	BNE,BNZ
1	1	1	1	=	F	B

Packed Arithmetic - SRP

- SS – Shift and Round Packed
- Used to shift packed fields to the left or right in order to compute decimal precision
- Op1 – Target Field (max 16 bytes)
- Op2 – A shift factor
  - Positive shift factor 1-31 – left shift
  - Negative shift factor in 2's complement – right shift
  - Express negative shift factor as 64 – n for n digit shift right
- Op3 – Rounding factor – (0-9)
  - 0 – no rounding
  - 5 – standard rounding
- Don't shift a significant digit off the left

SRP Examples

```
SRP  X,3,0 shift 3 digits left, no rounding
SRP  X,5,0 shift 5 digits left, no rounding
SRP  X,64-1,5 shift 1 right, round on 5
SRP  X,64-3,5 shift 3 right, round on 5
SRP  X,64-5,0 shift 5 right, no rounding

64-3 = 61 Decimal = B’111101’ plain binary
000011 = +3
111100 = complement
    +1
111101 = 2’s complement = -3
111101 = 63 in decimal
```

SRP Details

- Can shift up to 31 digits to the left, 32 digits to the right
- Shifting a non-zero digit off the left causes an overflow
- A decimal overflow exception occurs if the decimal overflow mask is 1, otherwise you can test for overflow with BO, BNO
- SPM can be used to control the mask bit

Packed Arithmetic

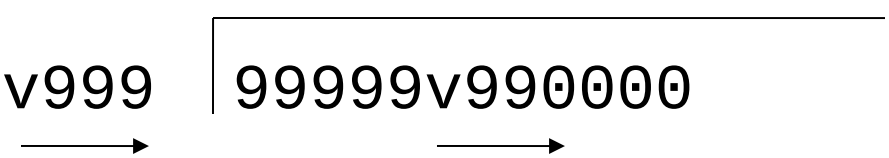
```
X     DC    P'-98765'
Y     DC    PL5'98765'
   SRP  X,64-3,5  X = 00099D

   SRP  Y,3,0     X = 098765000C

   SRP  Y,64-1,0  X = 000009876C

   SRP  X,1,0     ERROR or OVERFLOW
```

Shift Computations

```
X      DS  PL4  2 DECIMALS
Y      DS  PL4  3 DECIMALS
WORK   DS  PL10

Compute  X + Y to 2 decimals rounded

         ZAP  WORK,X
         SRP  WORK,1,0
         AP   WORK,Y
         SRP  WORK,64-1,5
```

Shift Computations

```
X      DS   PL4   2 DECIMALS
Y      DS   PL4   3 DECIMALS
WORK   DS   PL10
Compute  X / Y to 2 decimals rounded
```




```
         ZAP   WORK,X
         SRP   WORK,4,0
         DP    WORK,Y
         SRP   QUOT,64-1,5
```

Shift Computations

```
X       DS   PL4   4 DECIMALS
WORK    DS   PL8
COUNT   DS   P'9'

Compute  X^10 TO 4 decimals rounded

        ZAP   WORK,X
LOOP    EQU   *
        MP    WORK,X
        SRP   WORK,64-4,5
        SP    COUNT,=P'1'
        BNZ   LOOP
```

Converting from Packed to Numeric Edited Data

- Packed instruction - ED
- SS2
- Edit Pattern describes the output
  - X’40’ blank, fill pattern
  - X’20’ digit selector
  - X’21’ digit selector and significance starter
  - X’6B’ comma
  - X’4B’ decimal point

Edit Patterns

- An edit pattern is defined in hex and is called the edit “word”
- Packed field must match the edit “word”
- X’402020202120’ 5 digits
- X’4020202021204B2020’ 7 digits
- X’40202020’ 3 digits

Editing

- Create an edit word in memory that matches the packed field

```
XWORD    DC    X'40202120'
XPK      DS    PL2
```

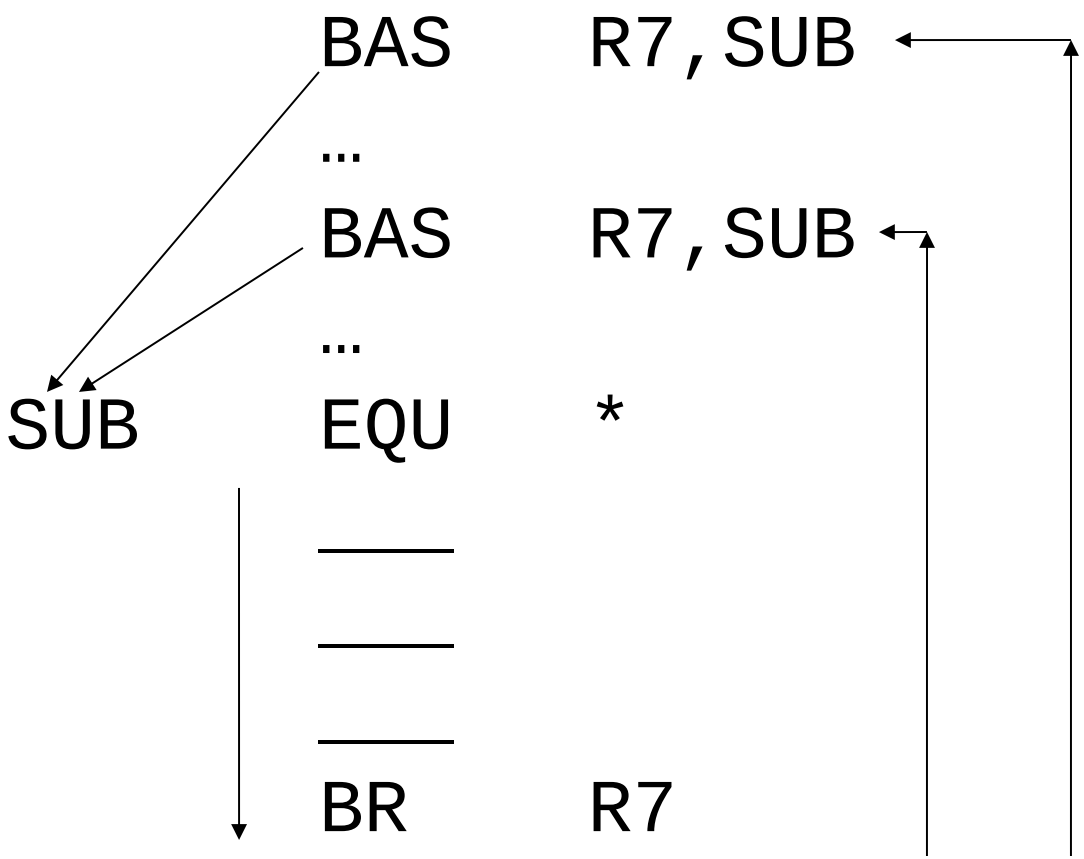
- Move the edit word to the output area (must match the edit word)

```
XOUT     DS    CL4
         MVC   XOUT,XWORD
```

- Edit the packed field into the output area

```
         ED    XOUT,XPK
```

Internal Subroutines



Practice Exercise #2

Greatest Common Divisor Computation for A and B

1. Let rem = remainder of dividing the larger number by the smaller number
2. Replace the larger number with rem
3. Stop if A or B = 0, print A + B. Otherwise go to step 1

Example	A	B	
	84	24	
	12	24	
	12	0	GCD = 12 + 0 = 12

Practice Exercise #2

- Create a file of records with two integers per record stored in a character format in columns 1-8
- Print each integer and the gcd. Print one record for each record in the input file

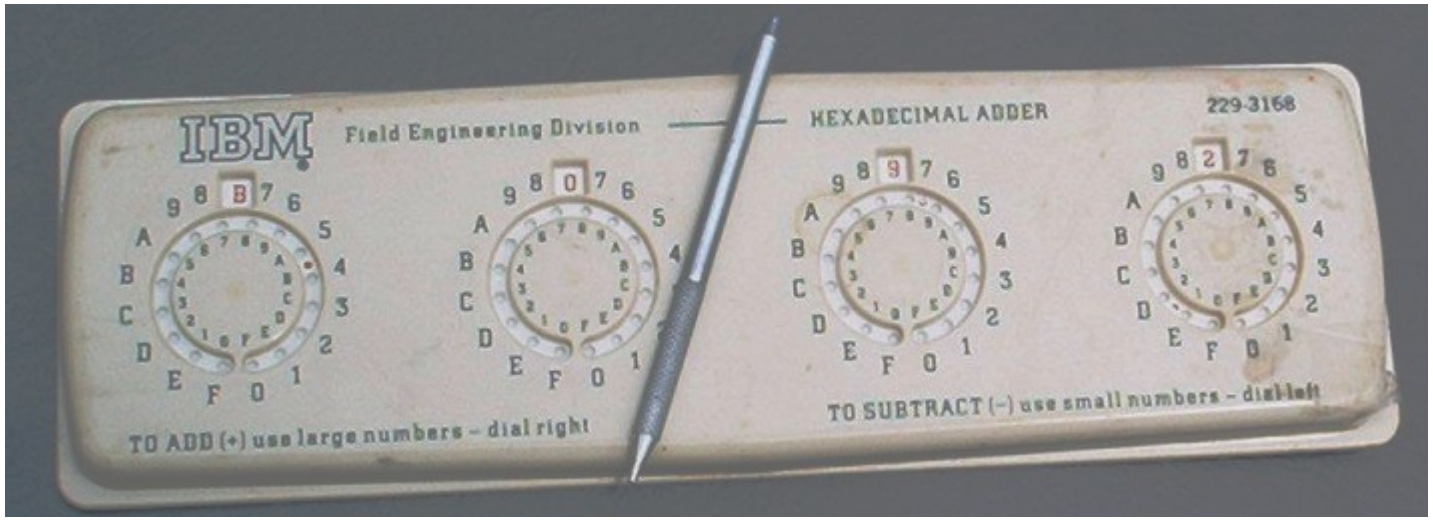
Practice Exercise #3

- Read a file that contains 3 5-byte fields in character format (cols 1-5,6-10, 11-15)
- Call the three fields A, B, C
- Assume A has 2 decimals, B has 3, and C has 4
- For each record in the file, print a record on a report
- Print A, B, C, A+B, (A+B)/C
- All answers should be good to one decimal rounded
- Don’t divide by 0

Binary Data

- Binary data is stored in 2’s complement format for signed arithmetic
- Binary data usually occurs as halfwords, fullwords, and doublewords
- Data can be defined as binary
- Data can be converted to binary from packed
- Some data is binary by nature (addresses)

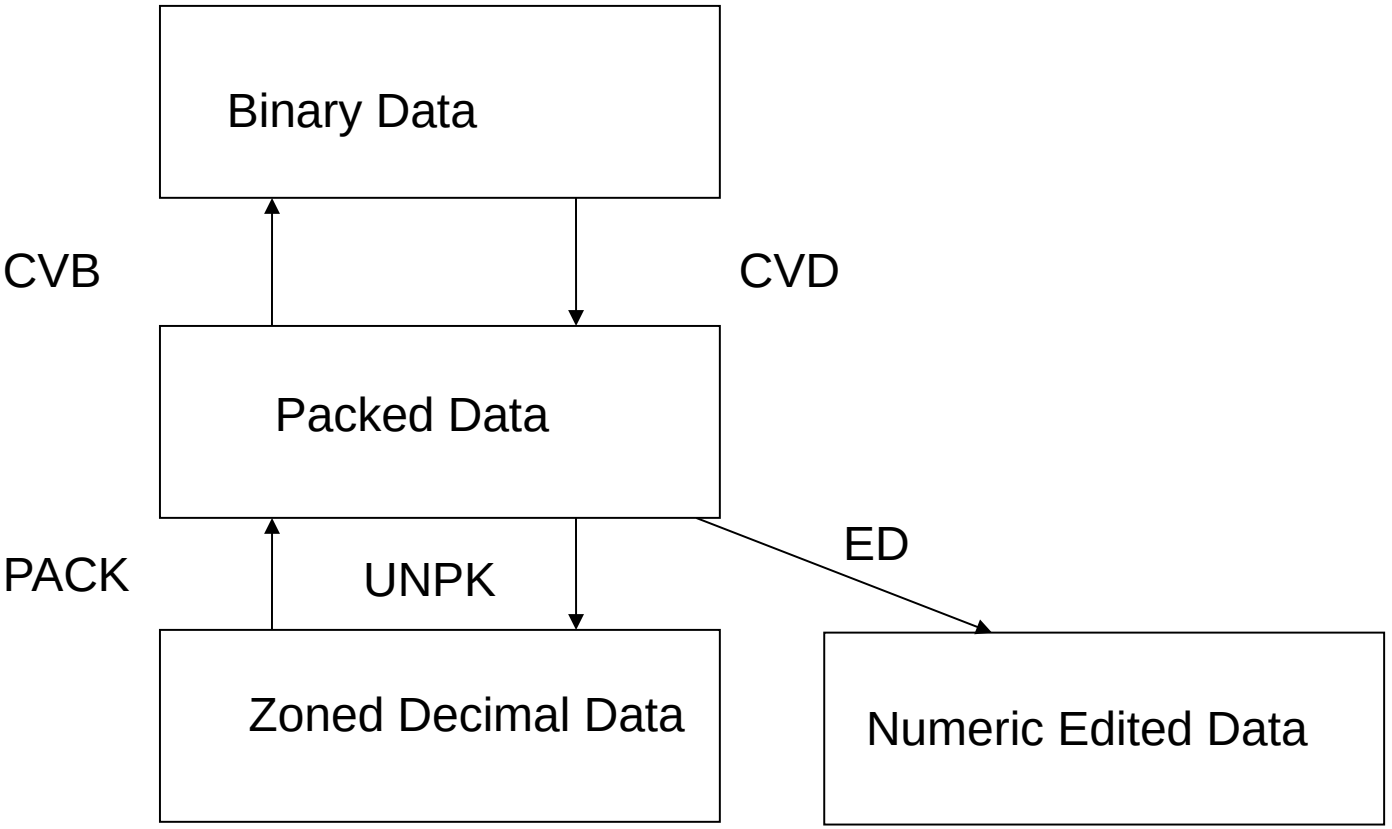
You might need one of these



Defining Binary Data

```
A     DC    F’1’              00000001
B     DC    H’-30’            FFE2
C     DC    D’50’ 0000000000000032
D     DC    B’10100000’ A0
E     DC    A(B)              ????????
F     DC    X’0040’           0040
```

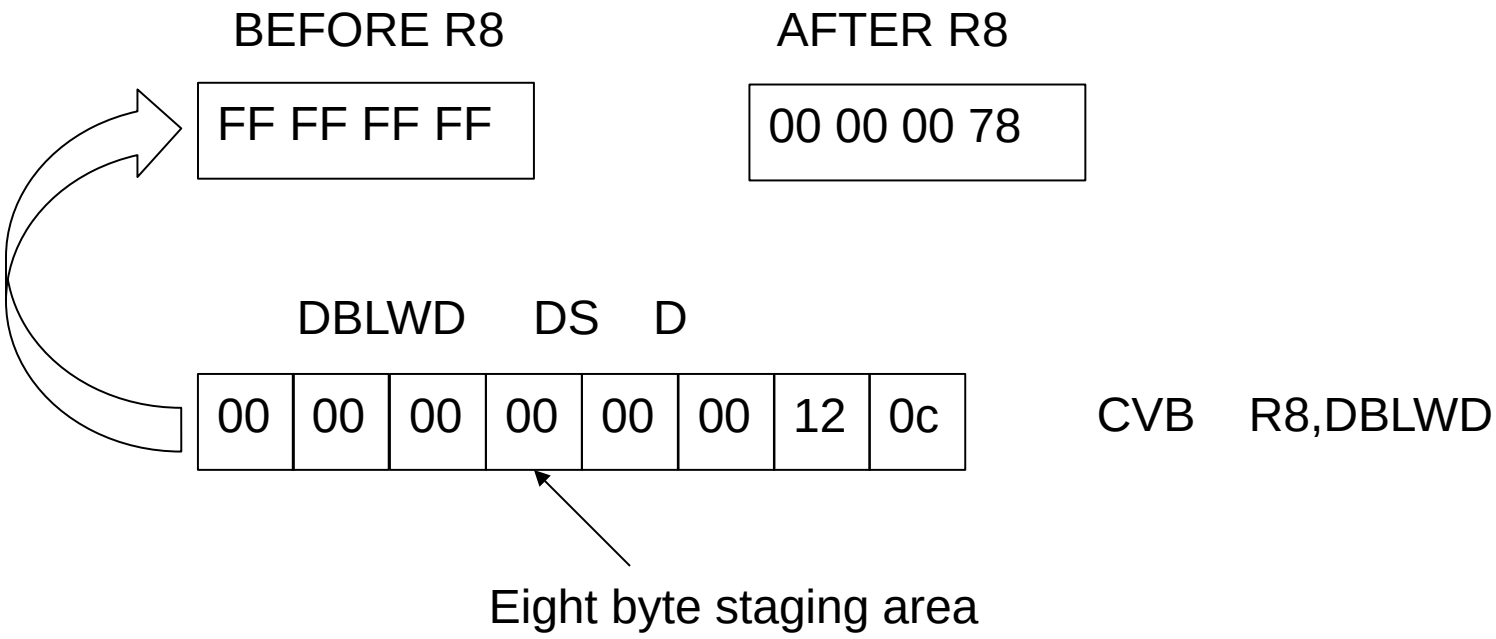
Converting Between Types



Convert to Binary

- CVB
- RX
- Operand 1 – Register – target
- Operand 2 – Doubleword in memory – contains packed data
- Packed data is converted to binary and placed in the register
- Some doubleword values will not fit in a register -2,147,483,648 -- +2,147,483,647
- Fixed point divide exception if conversion value out of range

Convert To Binary (CVB)



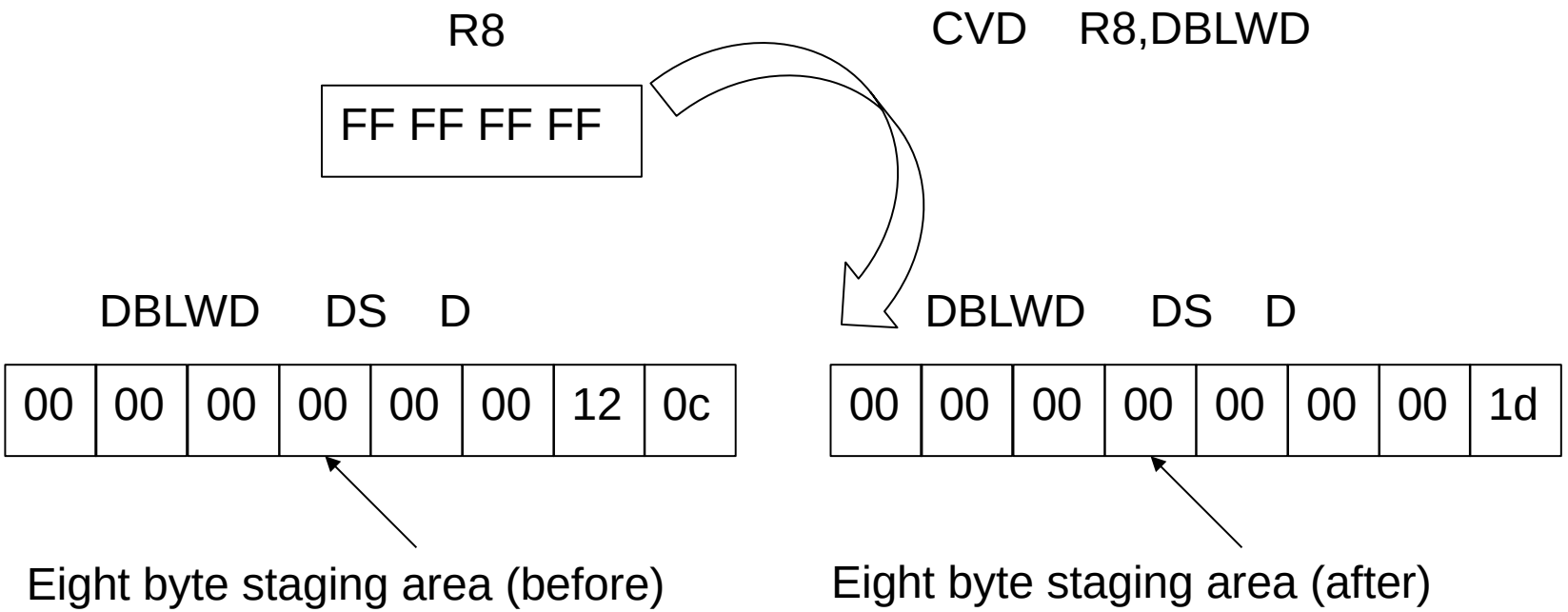
CVB Example

```
XPK      DC   P'70'
DBLWD    DS   D
              ZAP DBLDW,XPK
              CVB R8,DBLWD

     REG 8 = 00000046
```

Convert to Decimal

- CVD
- RX
- Operand 1 – Register - Source
- Operand 2 – Target is a Doubleword in memory – contains packed data afterward
- Binary data is converted to packed and placed in the doubleword

Convert To Decimal (CVD)



CVD Example

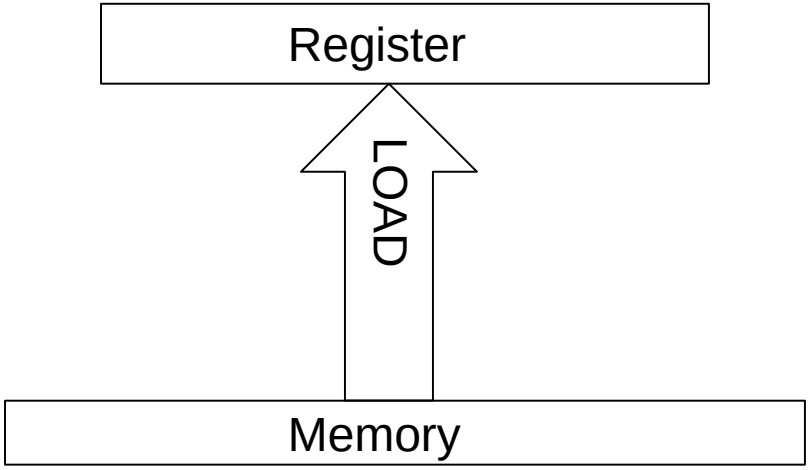
```
ASSUME R9 = 00000032
DBLWD    DS  D
XPK      DS  PL3
             CVD R9,DBLWD
             ZAP XPK,DBLWD
```

XPK = 00050C

Load Fullword

- RX
- L – Loads a binary fullword from memory into a register
- Op 1 – a Register
- Op 2 – a fullword in memory

```
X       DC      F'20'

        L       R8,X
```



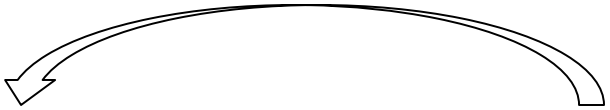
Load Register

- RR
- LR – Load Register
- Copies the contents of Op-2 into Op-1
- Op-1 – a target register
- Op-2 – a source register

Load Register Example

LR R3,R7

R3 (Before) R7 (Before)



00 00 11 22

44 33 22 11

R3 (After) R7 (After)

44 33 22 11

44 33 22 11

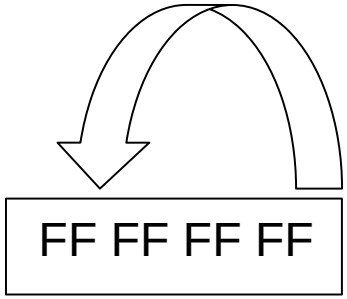
Load and Test Register

- RR
- Identical to LR but also sets the condition code to indicate how Op1 (final value) compares to zero
- Op-1 – a target register
- Op-2 – a source register

Load and Test Register Example

LTR R7,R7

R7 (Before) CC = 00 EQUAL



R7 (After) CC = 01 LOW

FF FF FF FF

Load Halfword

- RX
- LH – Loads a binary fullword into a register from a halfword in memory
- The halfword sign is propagated to fill the leftmost bytes of the register
- Op 1 – a Register
- Op 2 – a halfword in memory

LH

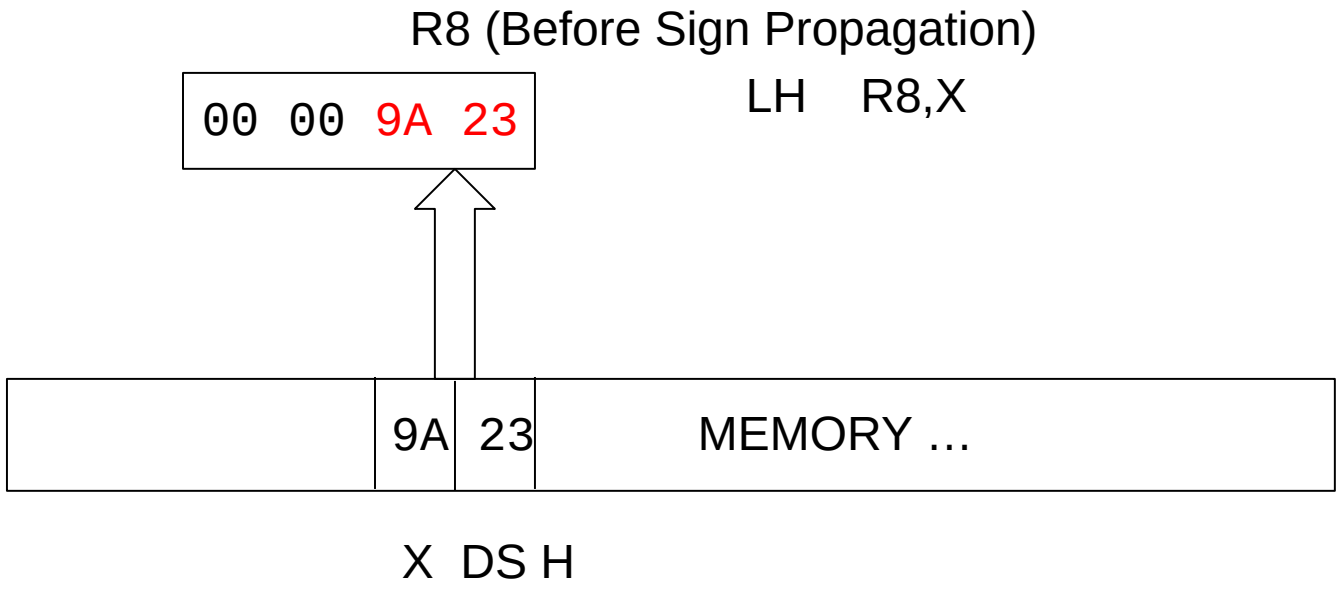
R8 (Before)

00	00	FF	FF

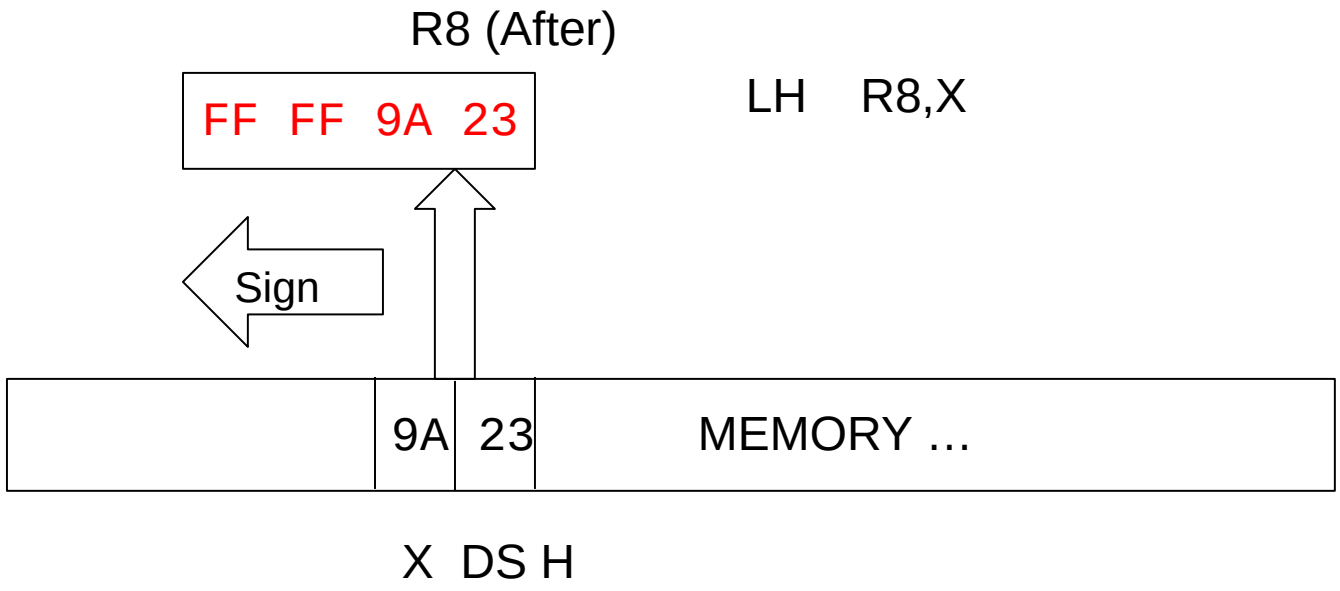
LH R8,X

X DS H

LH


R8 (Before Sign Propagation)
00 00 9A 23
LH R8,X
9A
23
MEMORY …
X DS H

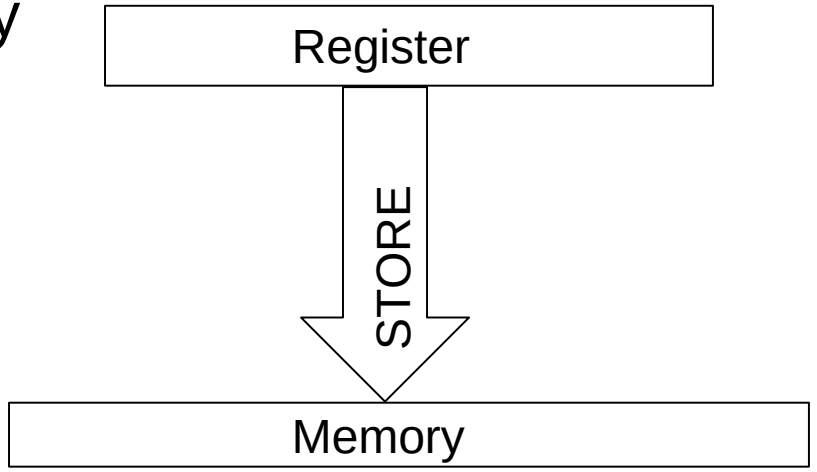
LH



Store Fullword

- RX
- ST – Copies the contents of a register into a binary fullword in memory
- Op 1 – a Register
- Op 2 – a fullword in memory

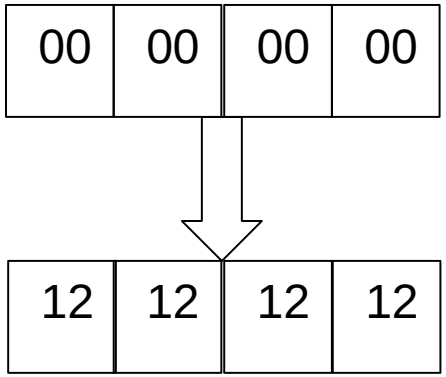
```
Y       DS      F

        ST      R4,Y
```



Store Fullword Example

ST R3,X

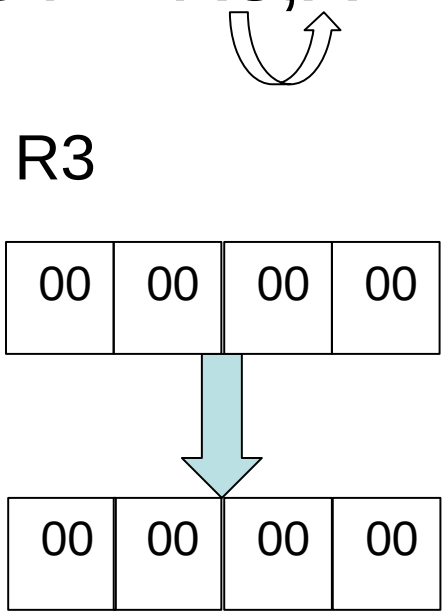
R3



X DS F (Before)

Store Fullword Example

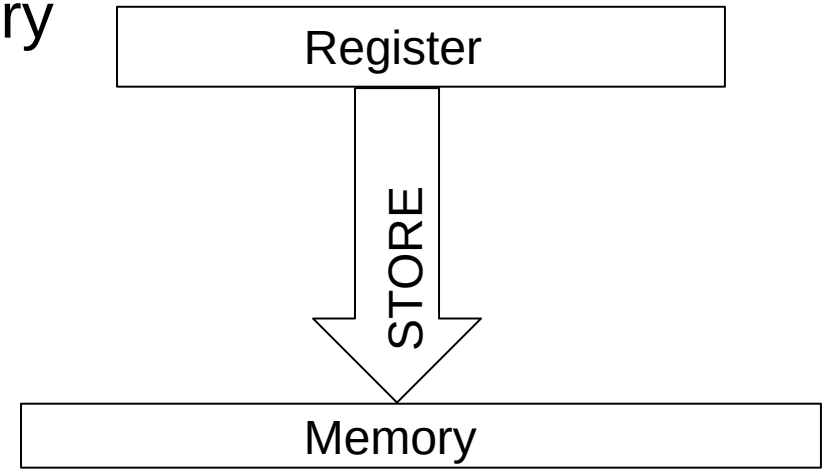
ST R3,X



Store Halfword

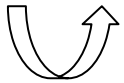
- RX
- ST – Copies the contents (two rightmost bytes) of a register into a binary halfword in memory
- Op 1 – a Register
- Op 2 – a halfword in memory

```
Y       DS      H

        STH     R4,Y
```

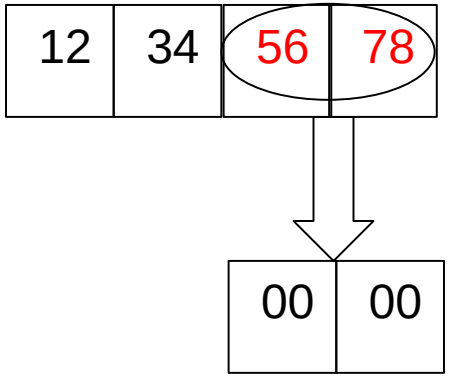


Store Halfword Example

STH R3,X



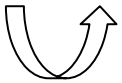
R3



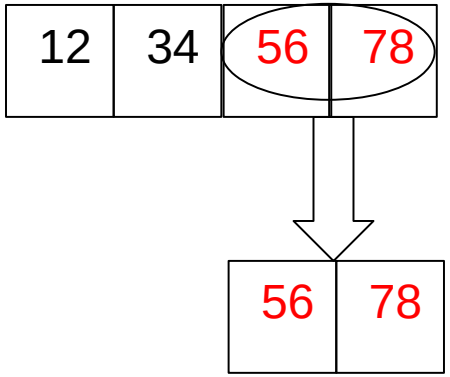
X DS H (Before)

Store Halfword Example

STH R3,X



R3



X DS H (After)

Putting Zoned Data in a Register

```
XZONED   DS    ZL5
XPK      DS    PL3
DBLWD    DS    D

         PACK  XPK,XZONED
         ZAP   DBLWD,XPK
         CVB   R7,DBLWD
```

Putting Packed Data in a Register

```
XPK      DS    PL3
DBLWD    DS    D

         ZAP   DBLWD,XPK
         CVB   R7,DBLWD
```

Putting Binary Data in a Register

```
XFULL      DC    F'-22'
XHALF      DC    H'32'

   L    R8,XFULL   R8 = FFFFFFEA
   LH   R9,XHALF   R9 = 00000020
```

RX and RR Instructions

Fullword	Halfword	Register
L	LH	LR
ST	STH	---
A	AH	AR
S	SH	SR
C	CH	CR
M	MH	MR
D	---	DR

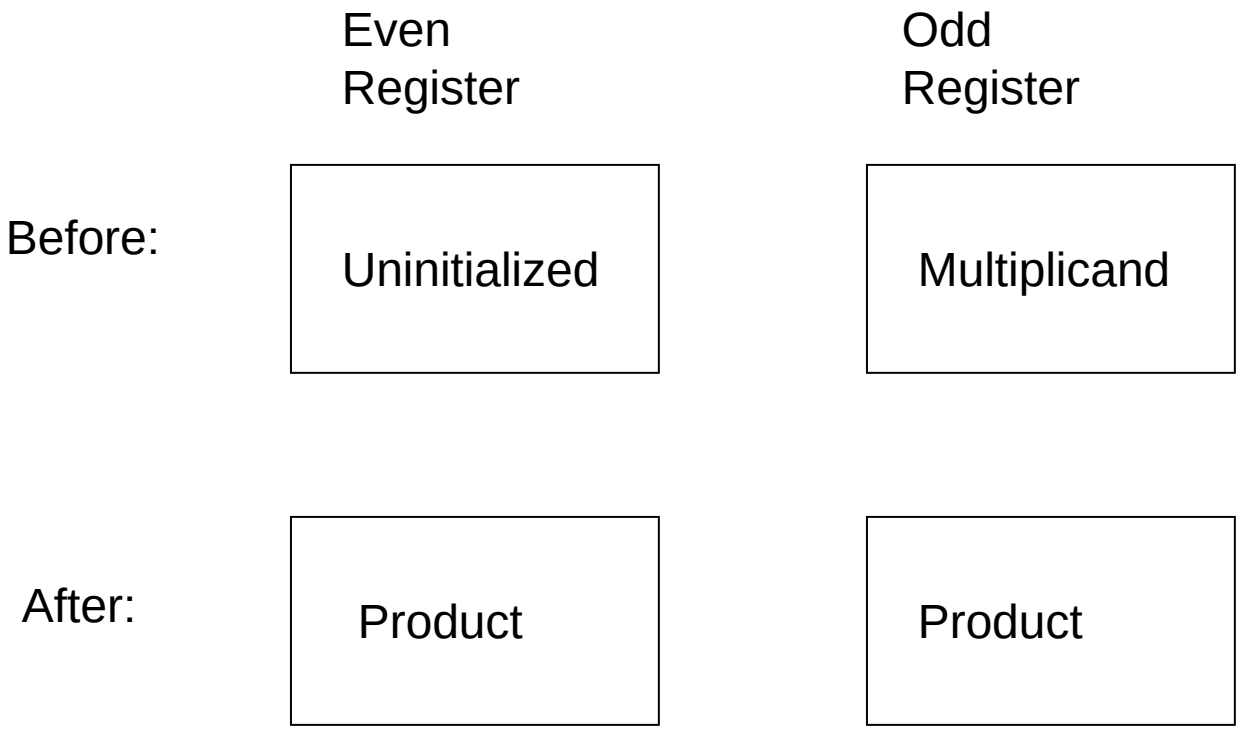
Add, Subtract, Compare Fullword

- RX
- A - Add Fullword
- S - Subtract Fullword
- C - Compare Fullword
- Operand 1 – Target Register
- Operand 2 – Fullword in memory (source)
- The fullword is added to, or subtracted from, the contents of the register
- For COMPARE, the contents are arithmetically compared

Add and Subtract Fullword, Halfword

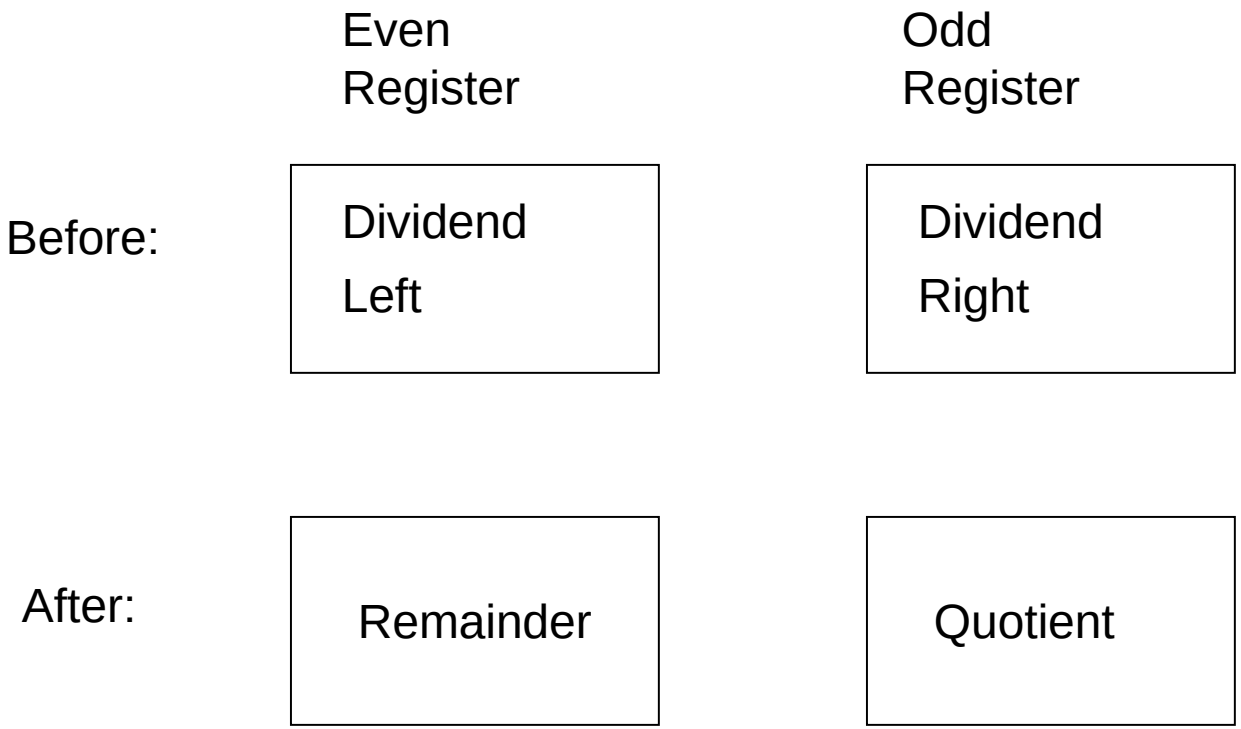
```
W       DS    F
X       DS    H
Y       DS    F
Z       DS    F
L       R8,W
AH      R8,X
S       R8,Y
ST      R8,Z
```

Multiply Fullword




```
X       DS      F
Y       DS      F
Z       DS      F
        L       R9,X
        M       R8,Y
        ST      R9,Z
```

Divide Fullword




```
X       DS    F
Y       DS    F
Z       DS    F
        L     R8,X
        SRDA  R8,32
        D     R8,Y
        ST    R9,Z
```

Add, Subtract, Compare Halfword

- RX
- AH - Add Halfword
- SH - Subtract Halfword
- CH - Compare Halfword
- Operand 1 – Target Register
- Operand 2 – The halfword is internally and temporarily sign-extended to a Fullword
- The Fullword is added to, or subtracted from, or compared to the contents of the register
- For COMPARE, the contents are arithmetically compared

Multiply Halfword

Register 7

Before:

After:

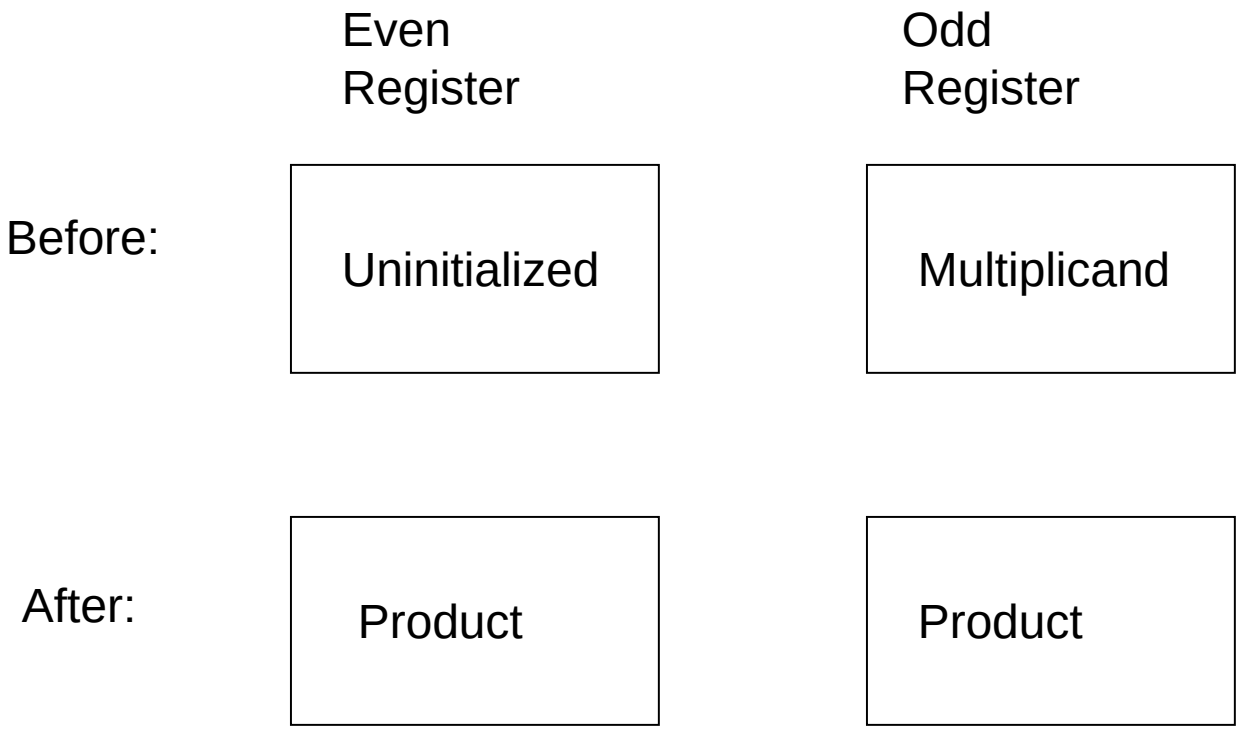

```
X       DS      F
Y       DS      H
Z       DS      F
        L       R7,X
        MH      R7,Y
        ST      R7,Z
```

Add and Subtract Register

- RR
- AR – Add Register
- SR – Subtract Register
- Op-1 – Target Register
- Op-2 – Source Register

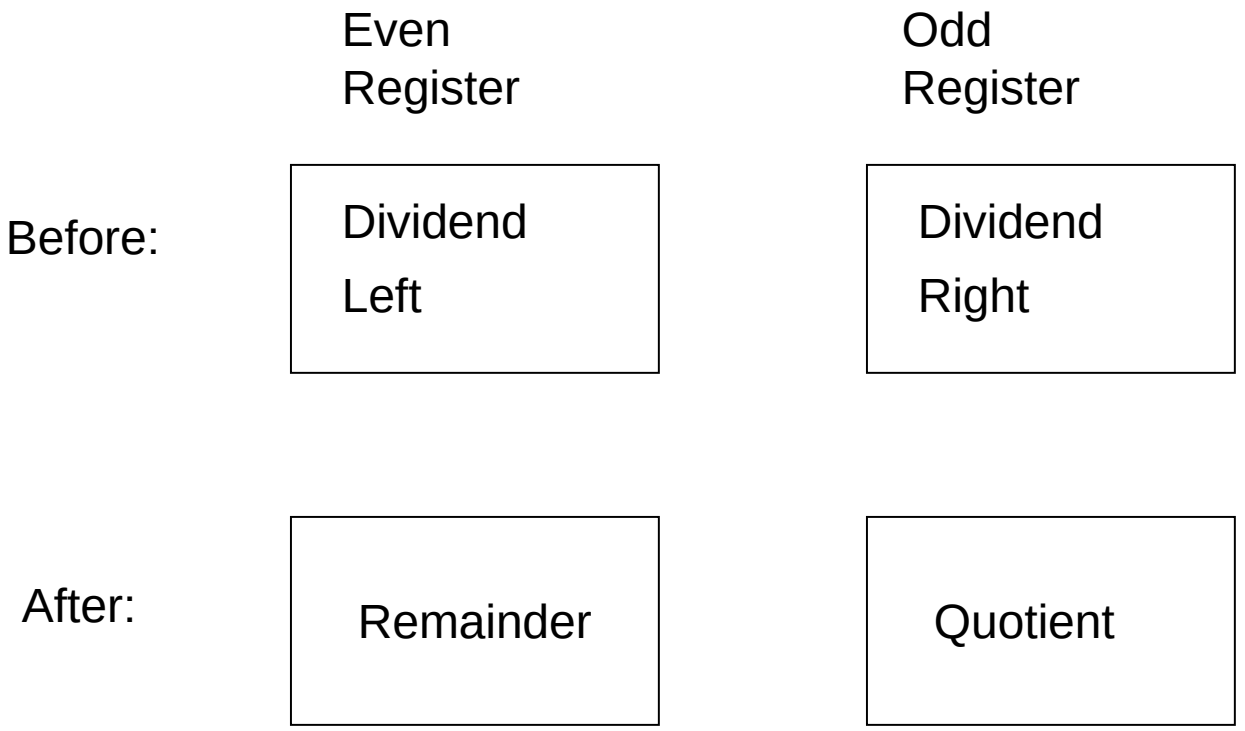
```
X   DS  F
Y   DS  F
    L   R8,X
    L   R9,Y
    AR  R8,R9
```

Multiply Register




```
X       DS      F
Y       DS      F
Z       DS      F
        L       R9,X
        L       R5,Y
        MR      R8,R5
        ST      R9,Z
```

Divide Register




```
X       DS      F
Y       DS      F
Z       DS      F
        L       R8,X
        L       R4,Y
        SRDA    R8,32
        DR      R8,R4
        ST      R9,Z
```

Alignment

- Doubleword, Fullword and Halfword alignment can be provided to a field:

```
          DS   0D   Doubleword alignment
X         DS  ….
          DS  0F    Fullword alignment
Y         DS  ….
          DS  0H    Halfword alignment
Z         DS  ….
```

Alignment

- Forced alignment can be prevented by coding a length for halfwords, fullwords and doublewords:

```
          DS   0F
X         DS   CL1
Y         DS   FL4
Z         DS   HL2
```

Practice Exercise 4

- Create a file of records
- Each record contains three integers
- Field A – columns 1-5 - zoned
- Field B – columns 6-7 – packed
- Field C – columns 8-11 – fullword
- Align the first byte of the record on a doubleword
- Print A, B, C, A+B, (A * B)/ C
- Do all the work as plain integer arithmetic in the registers

Load Address

- RX
- LA - Load address
- Op1 – the target register
- Op2 – a fullword in memory. The address of the memory location is copied into the register

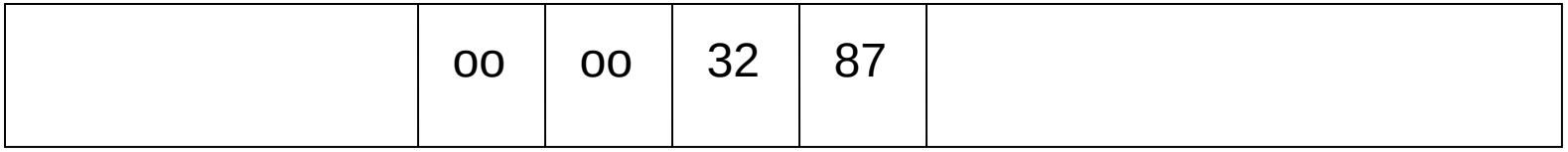
Load and Load Address

```
L       R8,X
LA      R9,X
```

After: R8 – 00003287

After R9 - 12FC0010

X



Byte 12FC0010

Branch on Count

- BCT
- RX
- Used to create counted loops with binary values
- Op 1 – a register containing a count
- Op 2 – a label of an instruction that is the target of a branch
- The register is decremented by 1. If the result is not zero, a branch occurs to the address indicated by Op 2. Otherwise, no branch occurs.

Counted Binary Loops

```
        L    R9,COUNT
        LA   R8,TABLE
LOOP    EQU   *
        MVC  NAMEO,NAME
        BCT  R9,LOOP
        …
COUNT   DC   F’30’
```

Branch on Count Register

- RX
- BCTR
- Op 1 – a register with a count field
- Op 2 – a register containing an address to branch to
- Op 1 is decremented, if the result is not zero, the branch occurs. Otherwise execution continues with the next instruction
- If Op 2 is specified as R0, no branch is taken, and execution continues with the next instruction

Edit and Mark

- EDMK
- Similar to Ed
- If significance starts because a significant digit was encountered, R1 points at the significant digit
- If the significance starter causes significance to start, R1 is unchanged

EDMK Example

```
EDWD     DC    X'402020202120'
AMTPK    DS    PL3
AMTO     DS   0CL6
         DS    CL5
AMTOLB   DS    CL1

         LA    R1,AMTOLB
         MVC   AMTO,EDWD
         EDMK  AMTO,AMTPK
         BCTR  R1,R0
         MVI   0(R1),C'-'
```

Typical Table

```
TABLE    EQU *
TABREC   DS 0CL8
QTYA     DS  F'90'
QTYB     DS  F'30'
RECEND   EQU *
         DC  F'30'
         DC  F'20'
         DS  F'66'
         DS  F'39'
TABEND   EQU *
RECLEN   DC  A(RECEND-TABLE)
ERECLEN  EQU RECEND - TABLE
TABLEN   DC  A(TABEND - TABLE)
ETABLEN  EQU TABEND - TABLE
NORECS   DC  A(ETABLEN/ERECLEN)
```

DSECTs

- DSECTs provide a technique for applying symbolic names to a storage area in memory
- A DSECT is a pattern of symbolic names and field descriptions
- The DSECT can be used to reference any storage area that can be addressed
- Uses: Table processing, parameter passing, Locate mode I/O
- NO storage associated with a DSECT

DSECT Creation and Use

```
CUST          DSECT
NAME          DS   CL20
ADDR1         DS   CL20
ADDR2         DS   CL20
CITY          DS   CL15
MAIN          CSECT
              …
              USING  CUST,R8
              LA     R8,TABLE
              MVC    NAMEO,NAME
              …
TABLE         DS   30CL75
              ORG  TABLE
TABREC        DS     CL75
RECLEN        EQU    *-TABLE
              DS   29CL75
```

“Bumping” a DSECT

```
         LA       R8,TABLE
LOOP     EQU   *
         MVC   NAMEO,NAME
         LA       R8,L’TABREC(R0,R8)
*        LA       R8,RECLEN(R0,R8)
         BCT    R9,LOOP
```

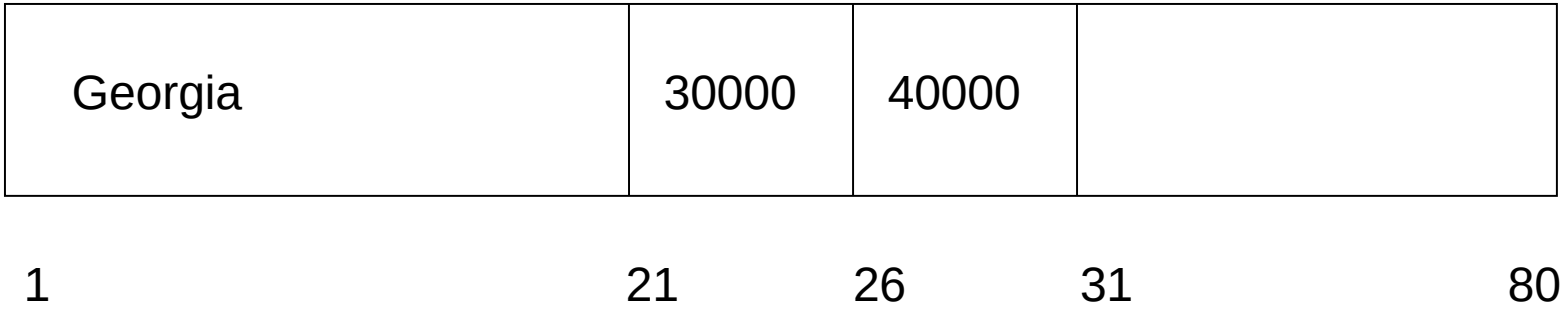
(Assumes TABREC DS CL75)

(Assumes RECLEN EQU *-Table)

Exercise #5

- Create a static table of states and zipcode ranges in your program
- Create and read a file of 5-digit zip codes (one zip per record)
- Search for each zip code in the given ranges of the table.
- Print out the corresponding state or a message indicating the zip is invalid

Exercise #5



Typical zip code table entry

Exercise #5A

- Use BCST.SICCC01.PDSLIB(EXER5A) which has 80 byte records in the following format:

  CUSTNO – cols 1-5 character

  CUSTBAL – cols 6-9 packed (dollars and cents)
- Read the data into a table in your program. After reading the entire file, process the table by printing a report that lists customer numbers and balances. Process the table again and print a second report that lists all the customer numbers with a negative balance.

Loading Multiple Base Registers

Programmer Responsibilities

- Tell the assembler which registers to choose when creating base/displacement addresses

  X DS CL5 ==> C008 (BDDD)

  Do this with USING:

  USING *,R12

- Load the base register with the correct address

Loading a Single Register

```
Address
X‘1000’    BASR      R12,R0
           USING    *,R12
X’1002’    ...
```

Why is this Incorrect?

```
Address
            USING   *,R12
X’1000’     BASR    R12,R0
X’1002’     ...
```

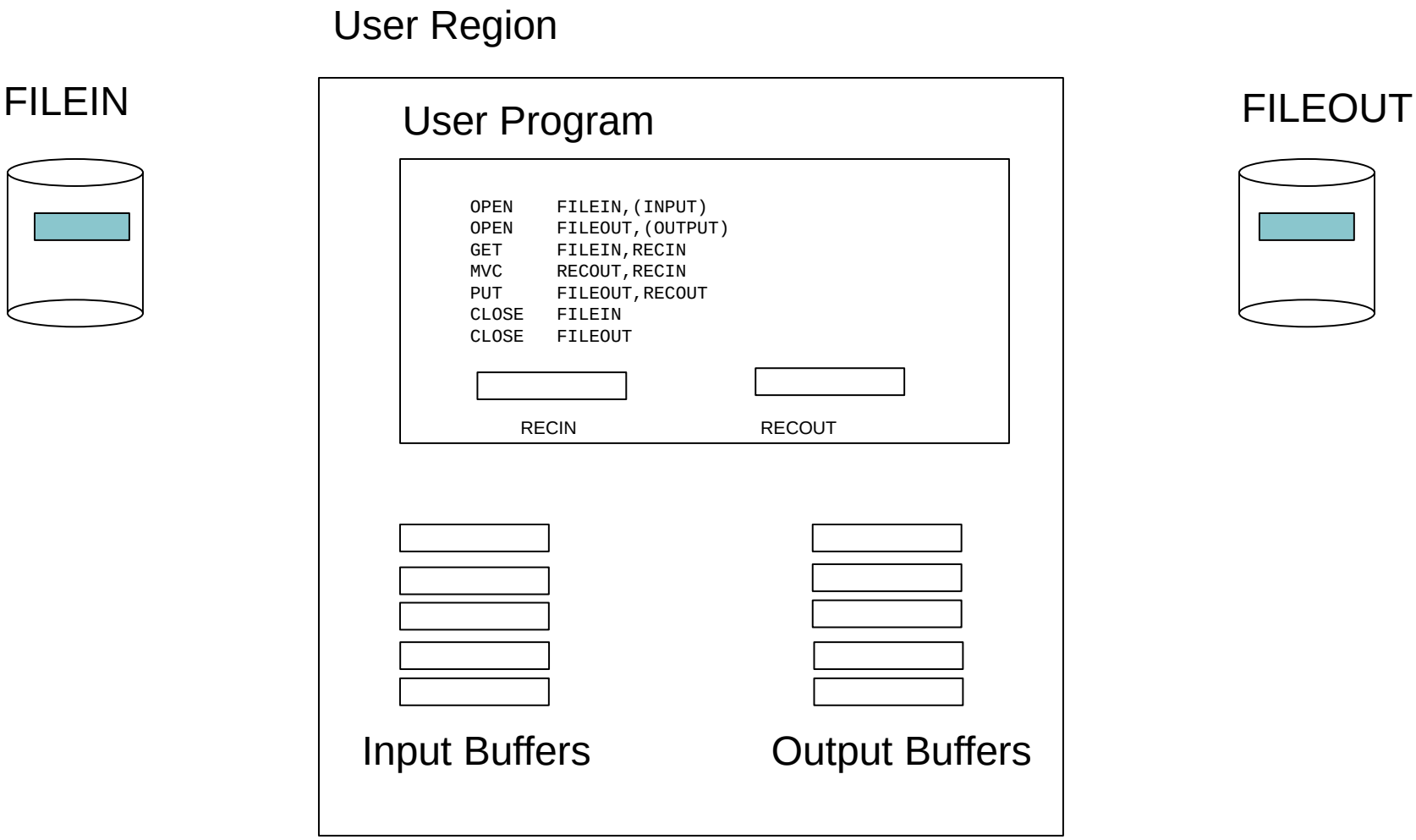
Loading Multiple Registers

```
Address
X‘1000’ BASR   R12,R0
        USING  *,R12,R11,R10
X’1002’ LA     R10,2048
        LA     R11,2048(R12,R10)
        LA     R10,2048(R11,R10)

Remember that X’1000’ = 4096
              X’800’  = 2048
```

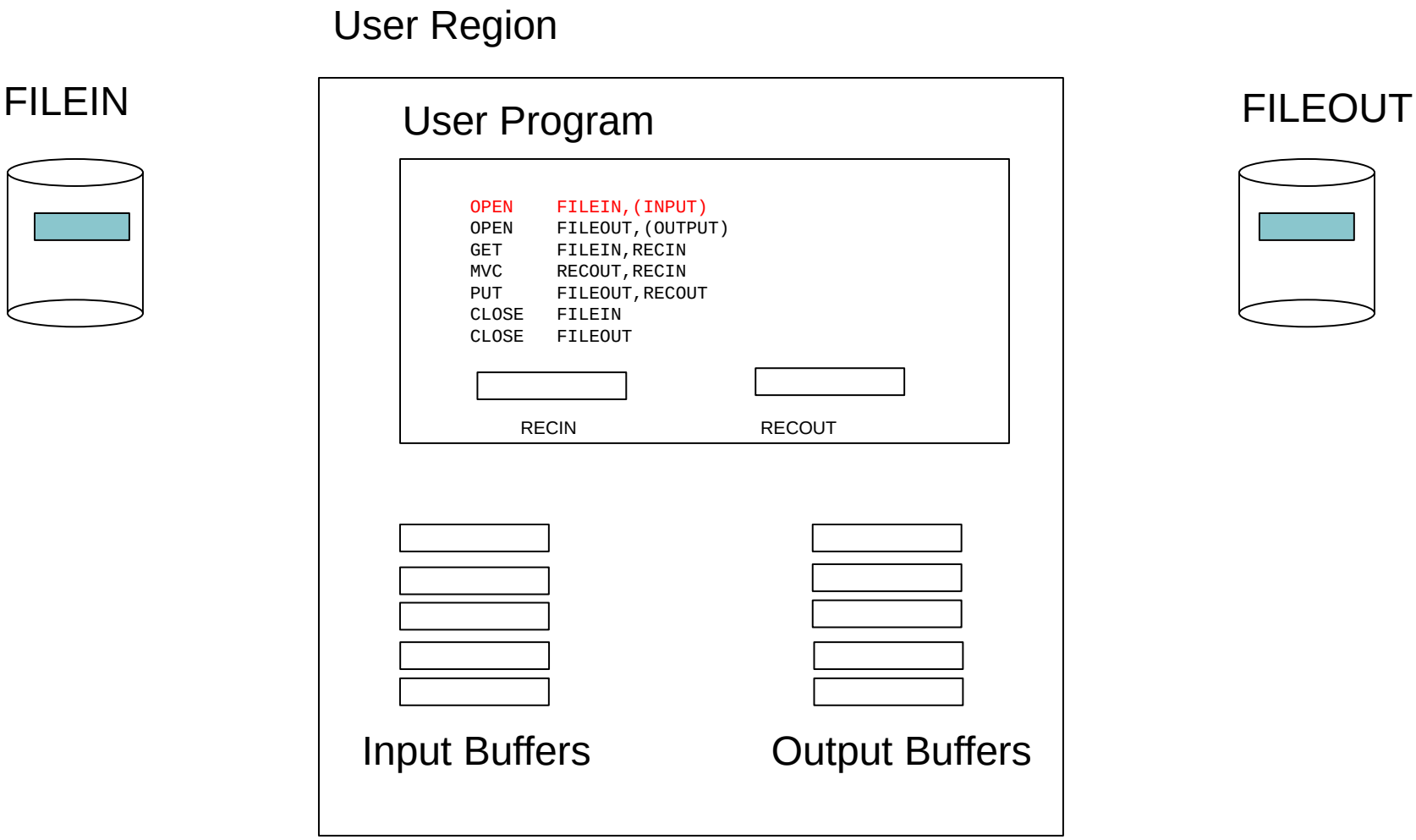
Direction

- It would be helpful to understand the terms **Domain** and **Range** as they relate to USINGs.
- Also, read about the **DROP** directive

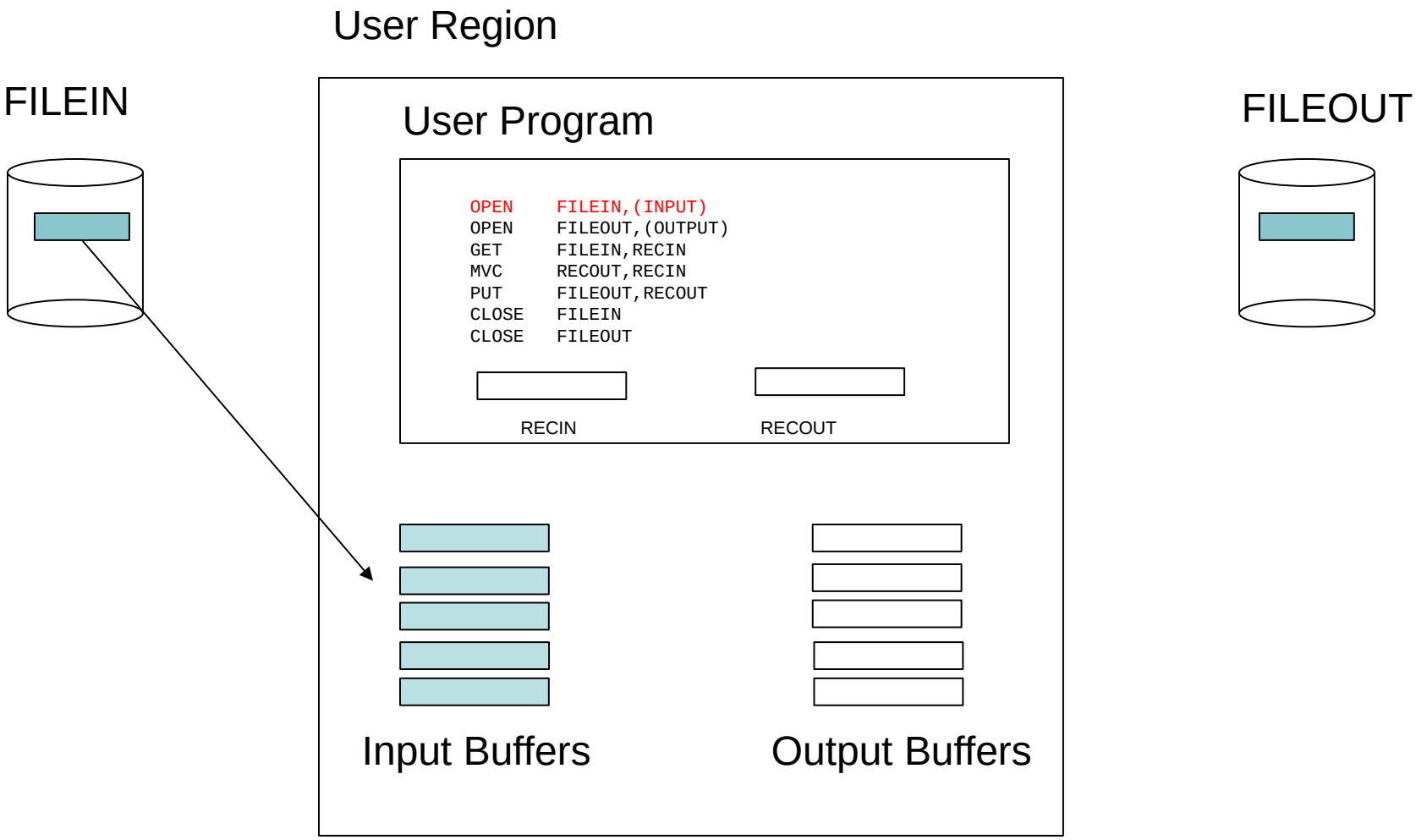
QSAM Move Mode I/O



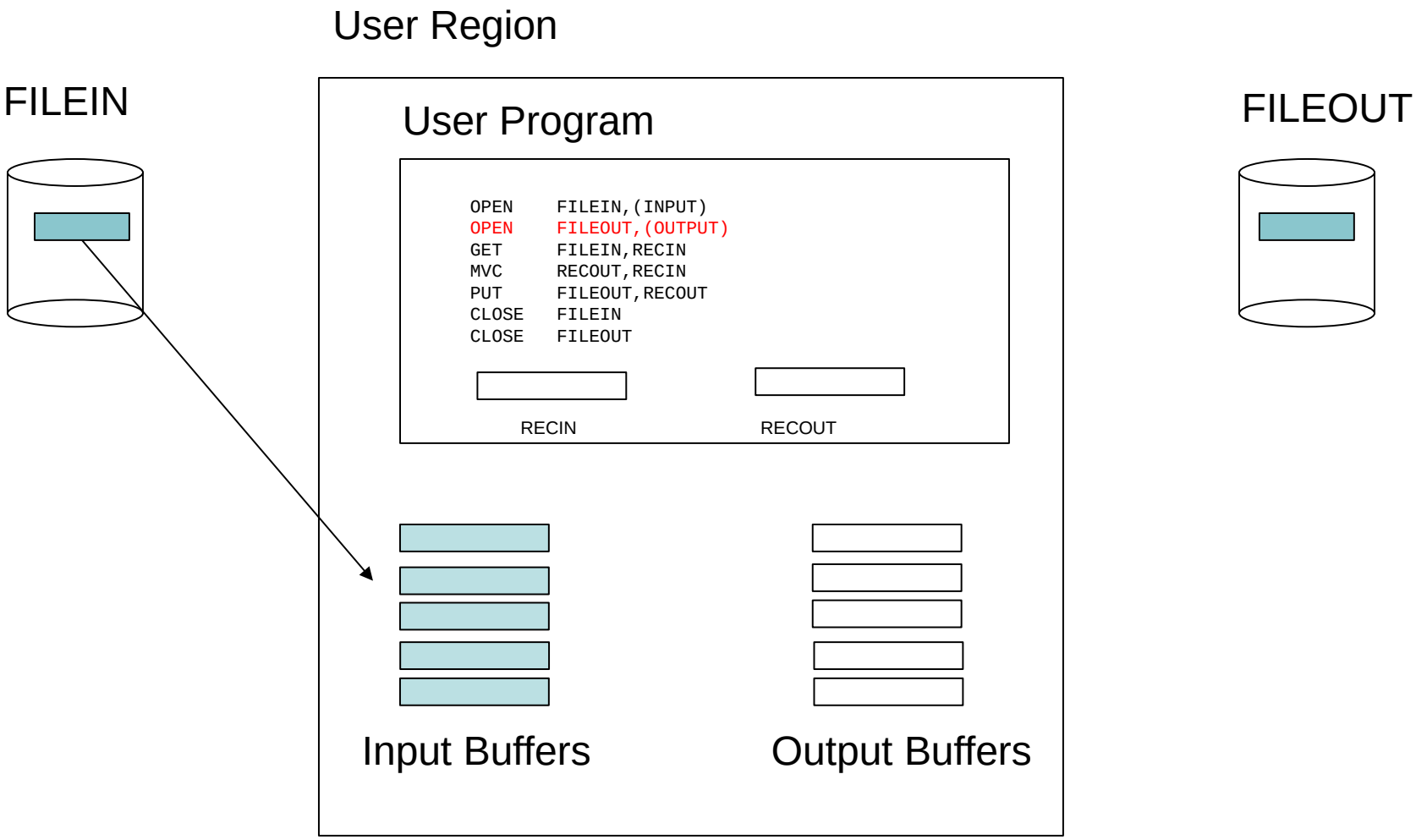
QSAM Move Mode I/O



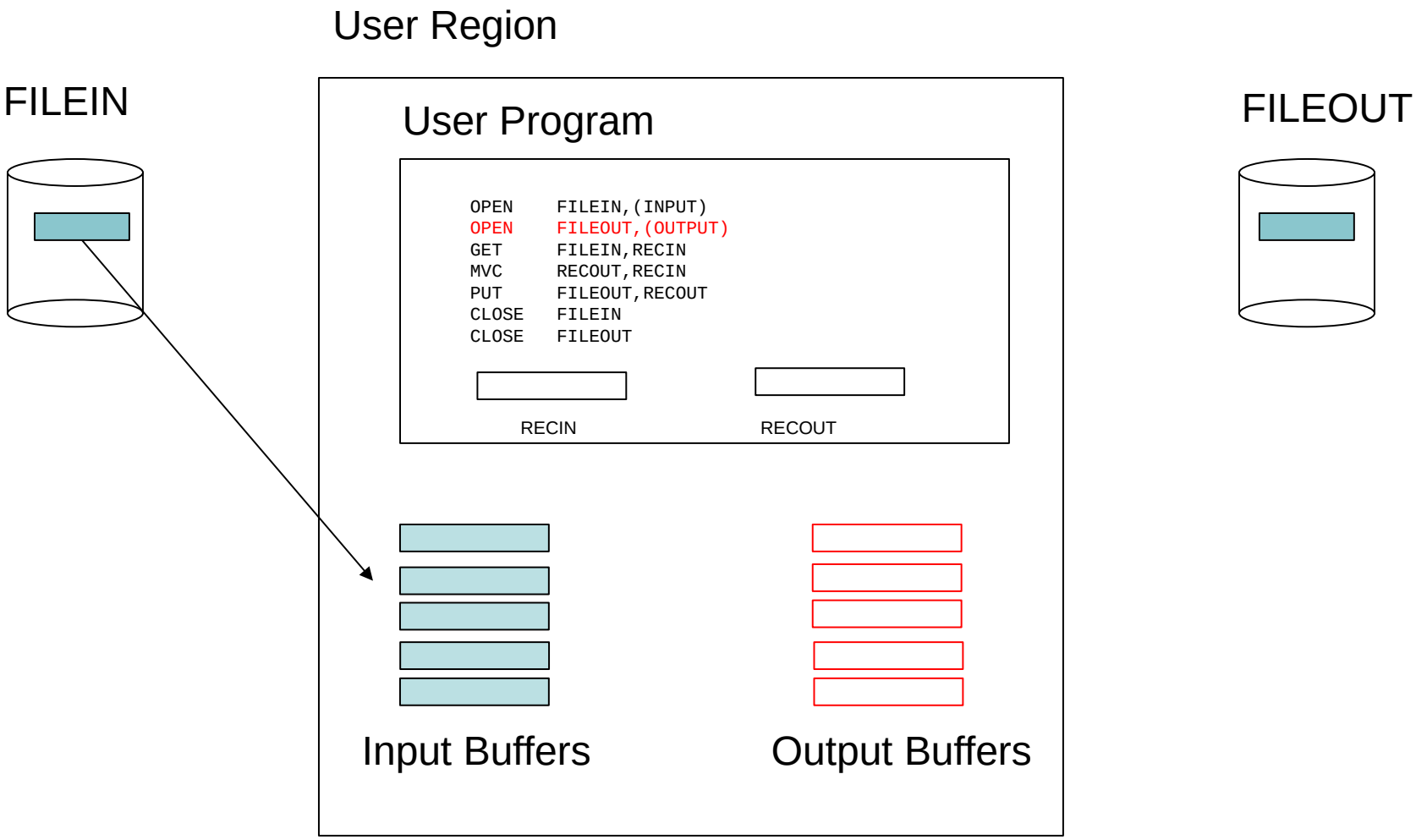
QSAM Move Mode I/O



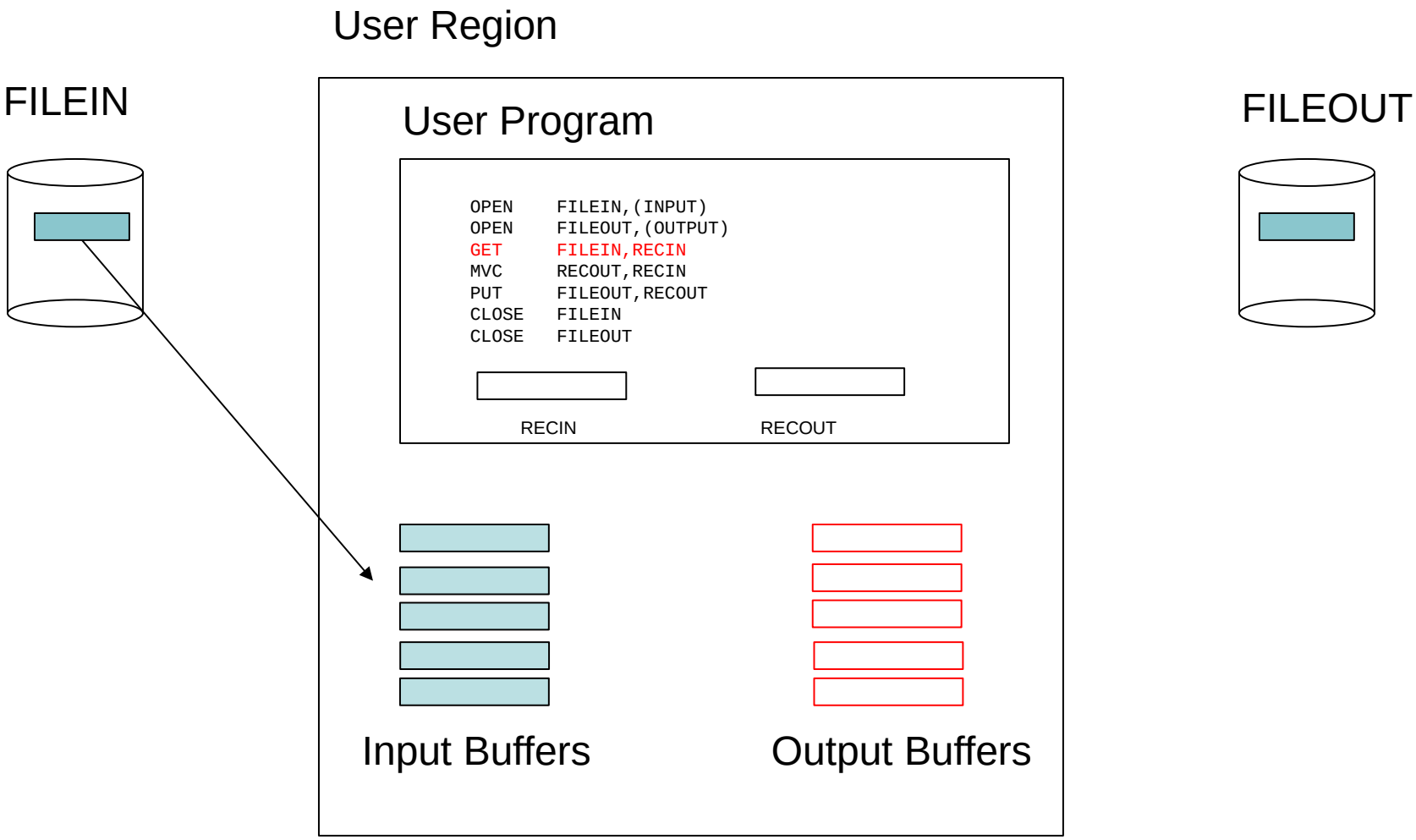
QSAM Move Mode I/O



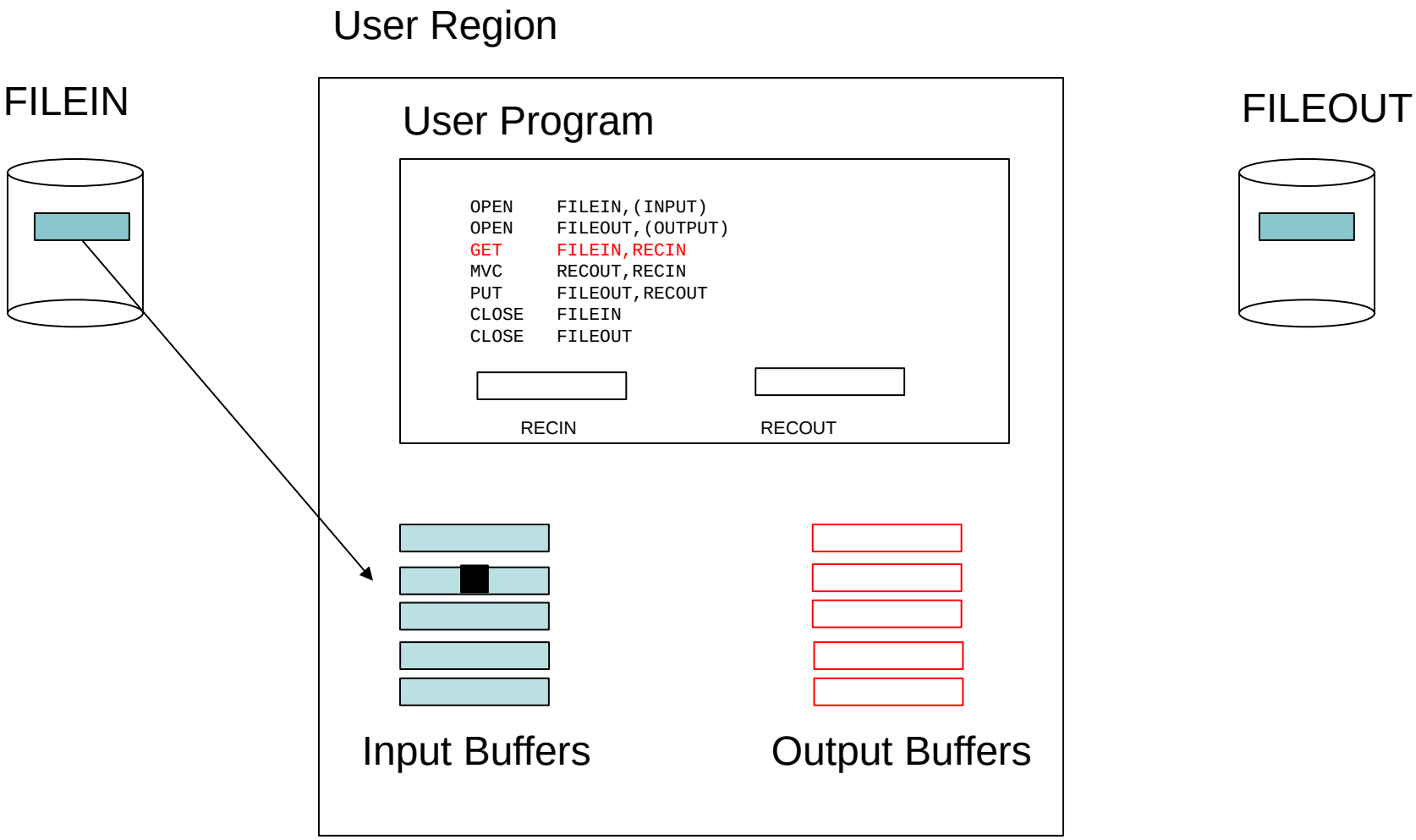
QSAM Move Mode I/O



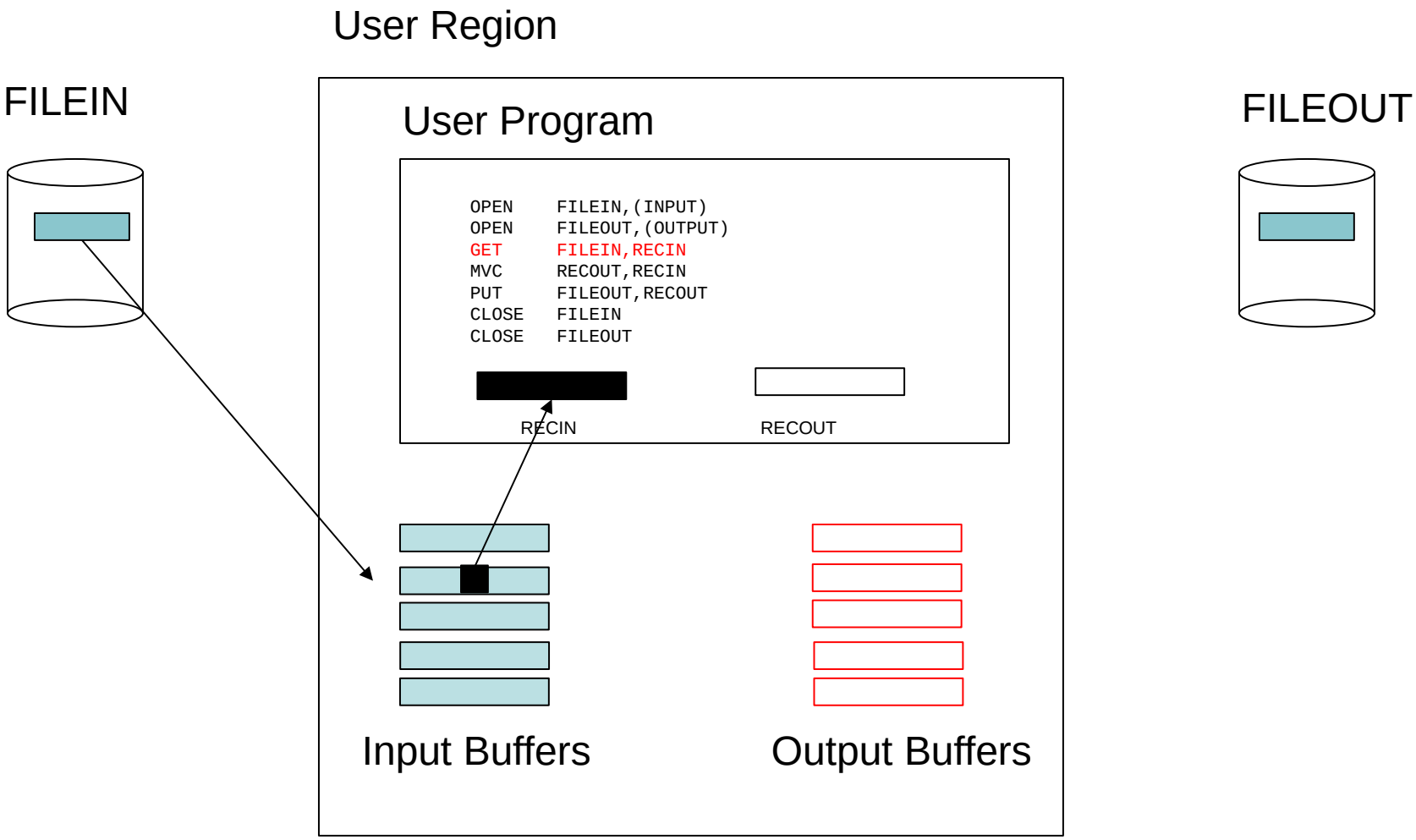
QSAM Move Mode I/O



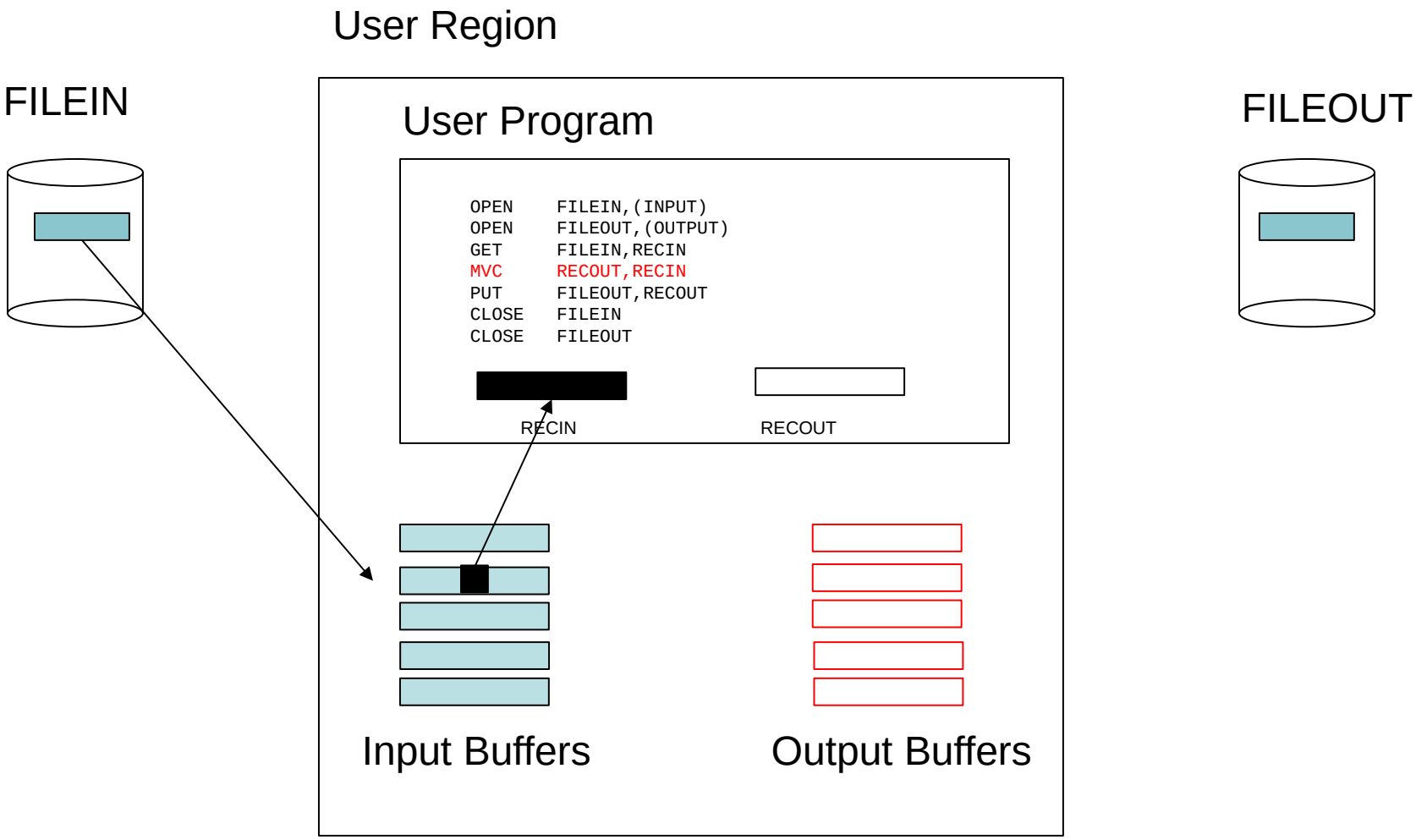
QSAM Move Mode I/O



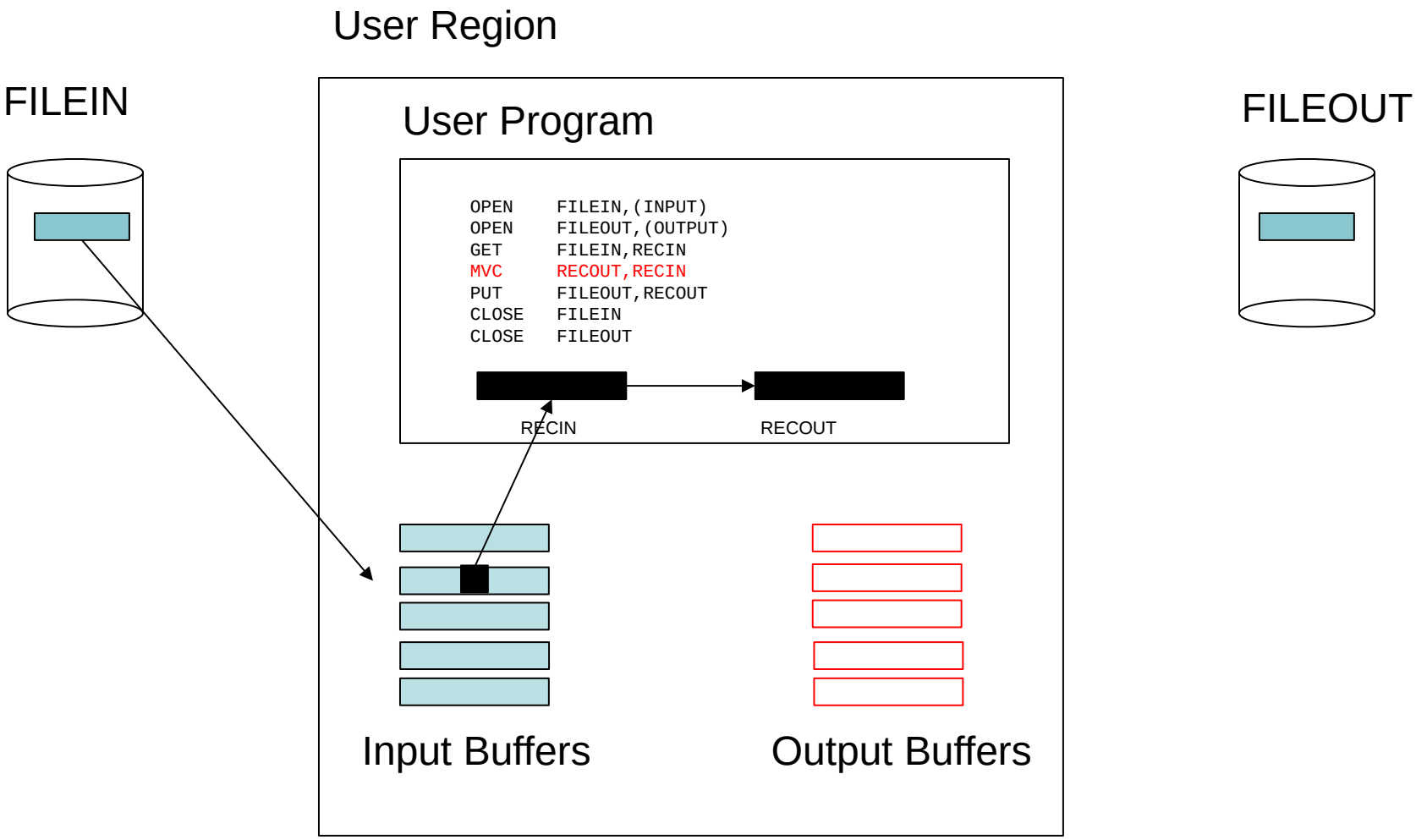
QSAM Move Mode I/O



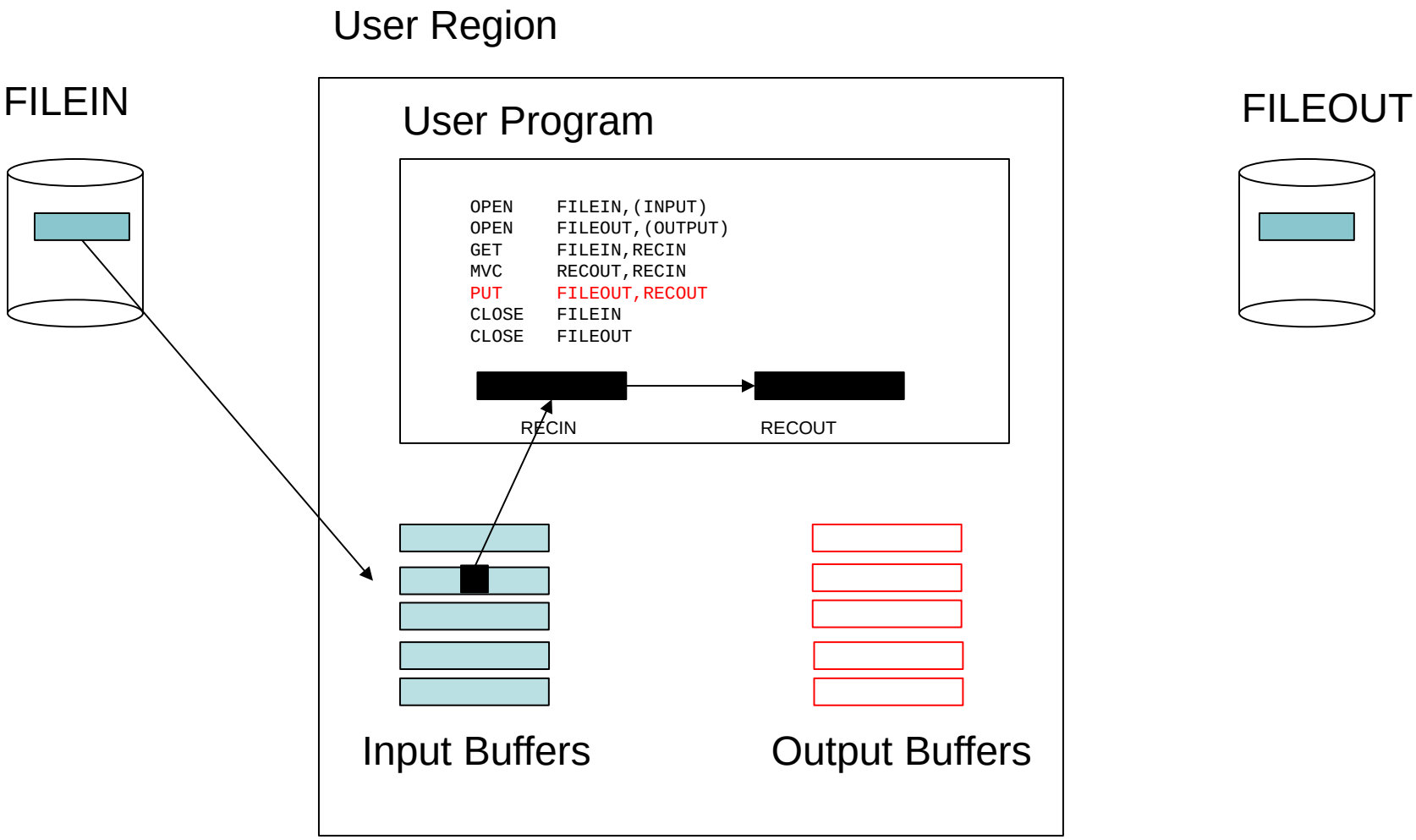
QSAM Move Mode I/O



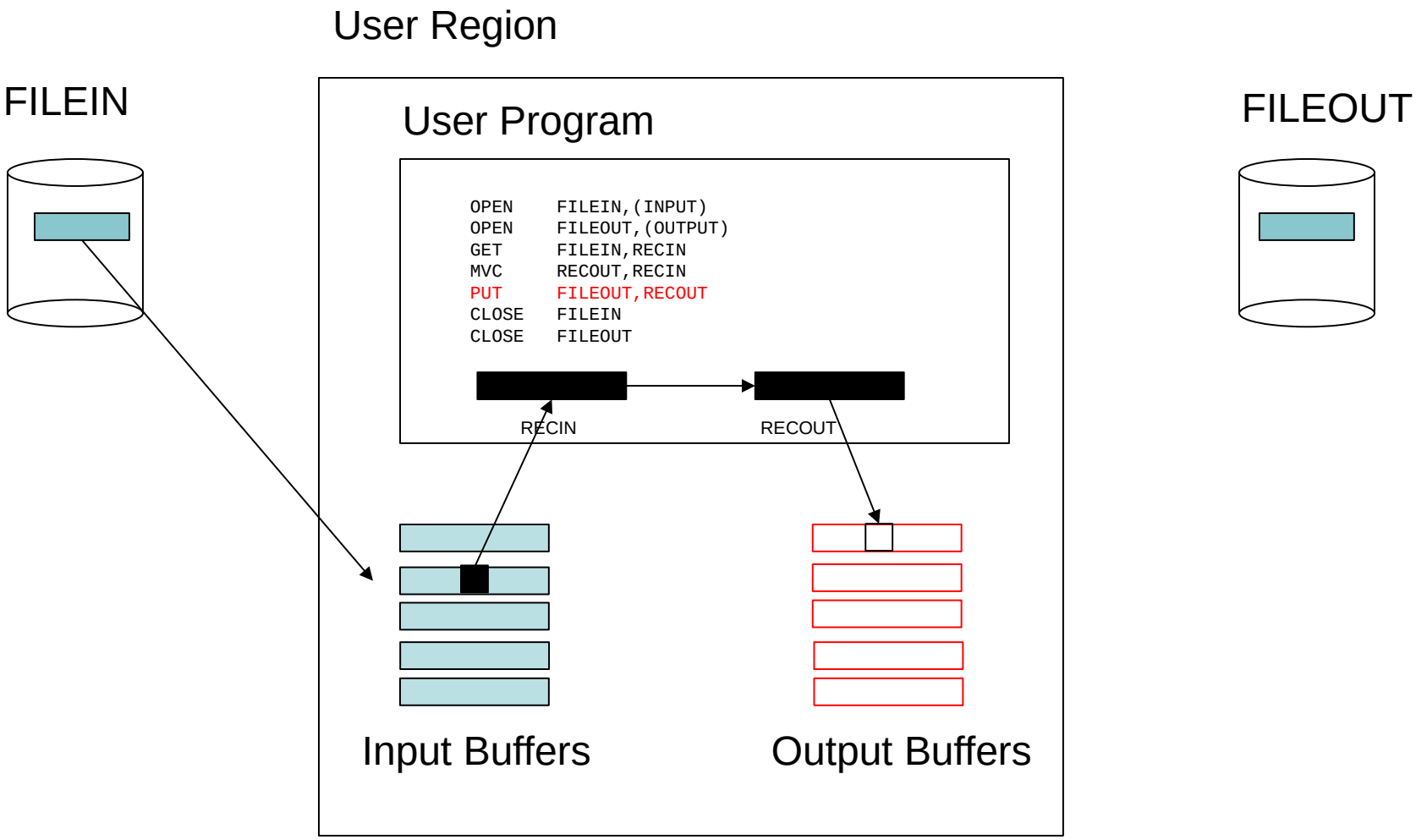
QSAM Move Mode I/O



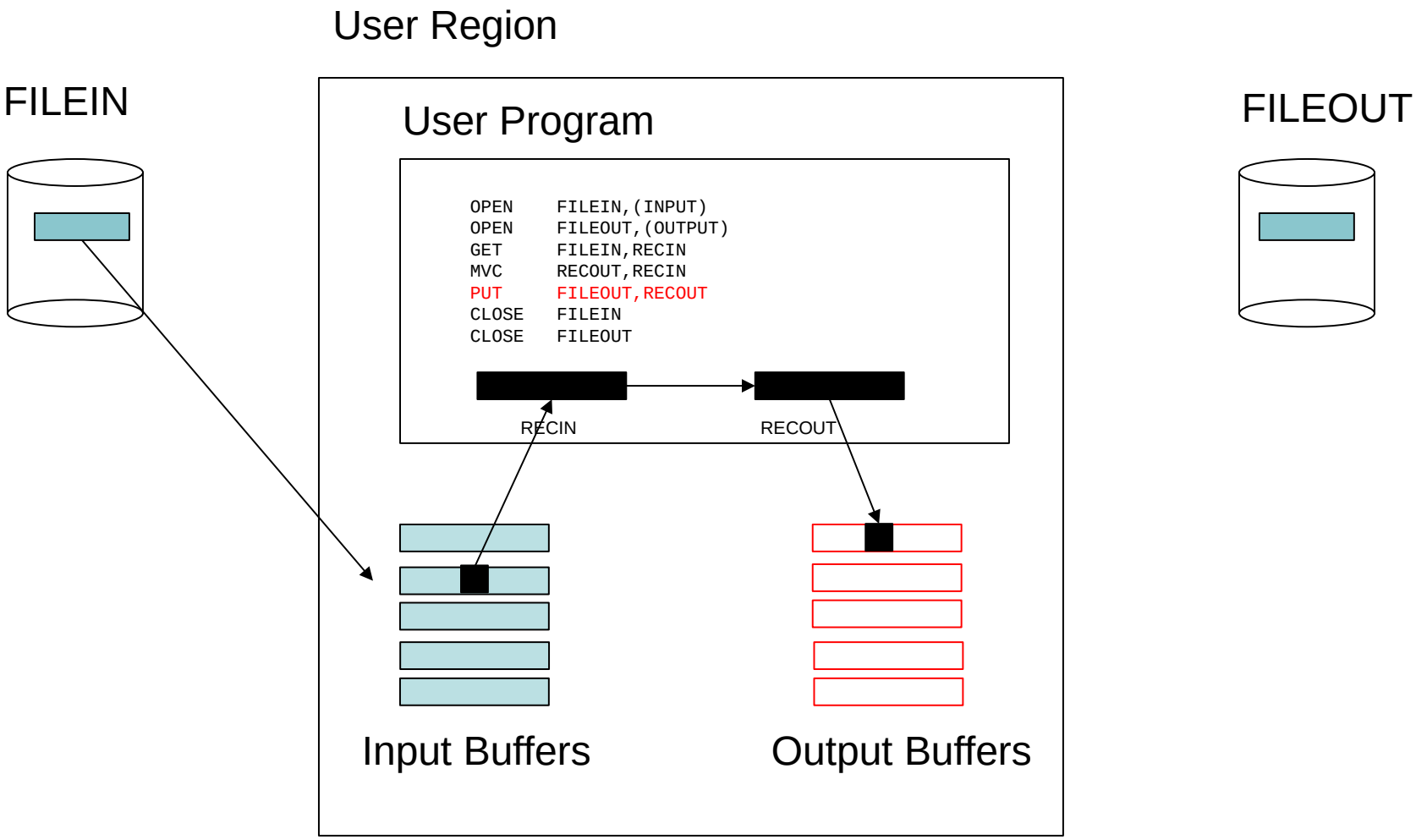
QSAM Move Mode I/O



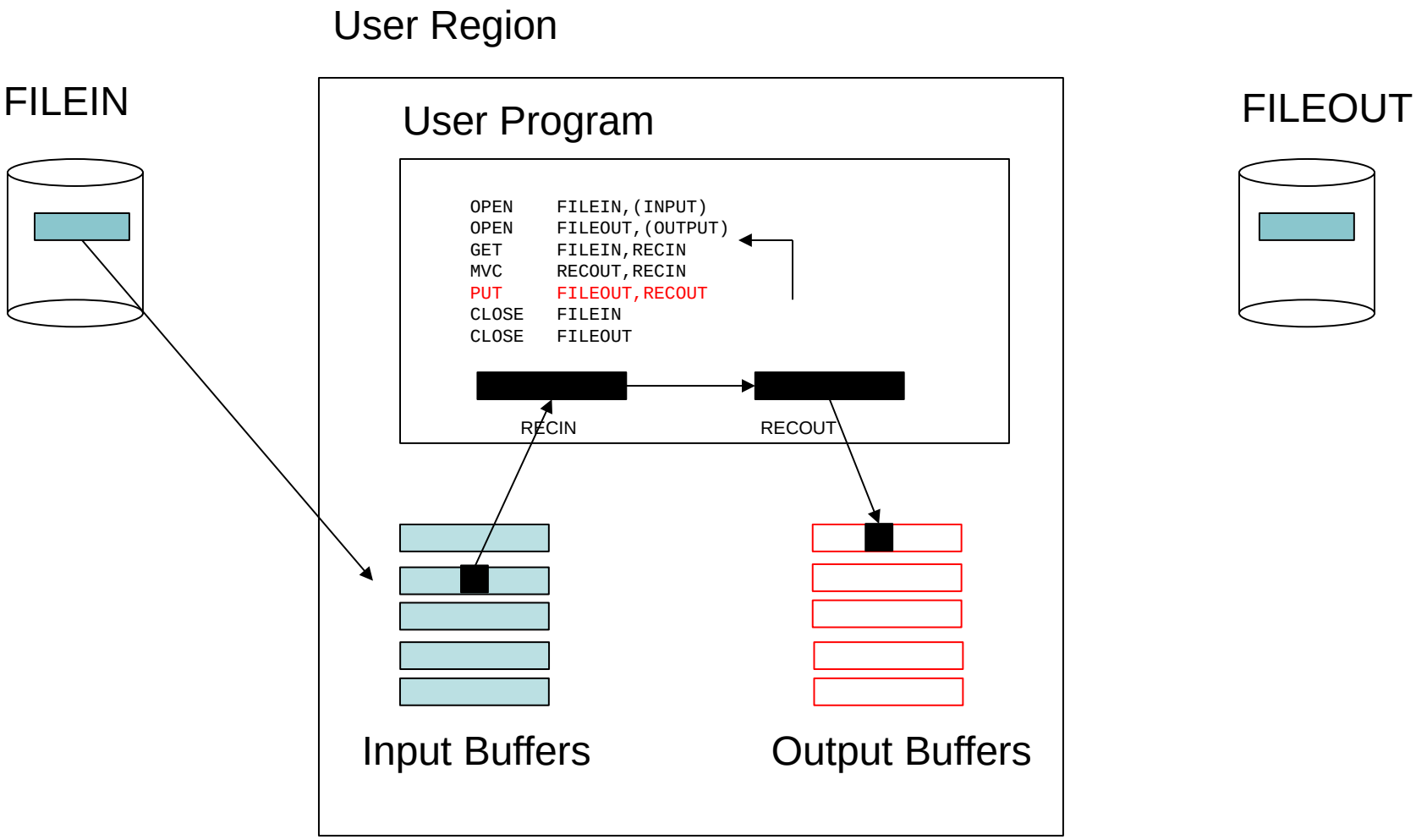
QSAM Move Mode I/O



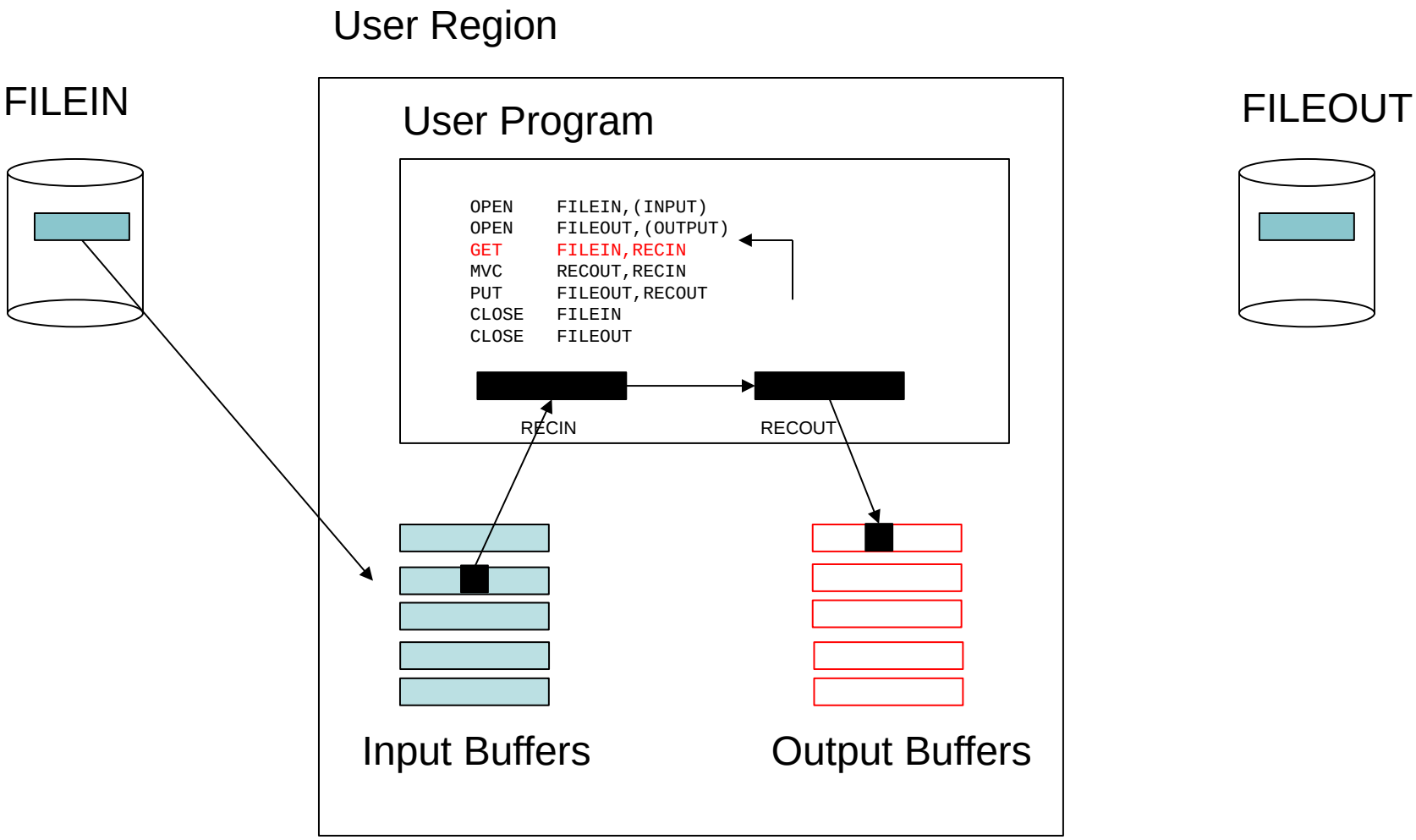
QSAM Move Mode I/O



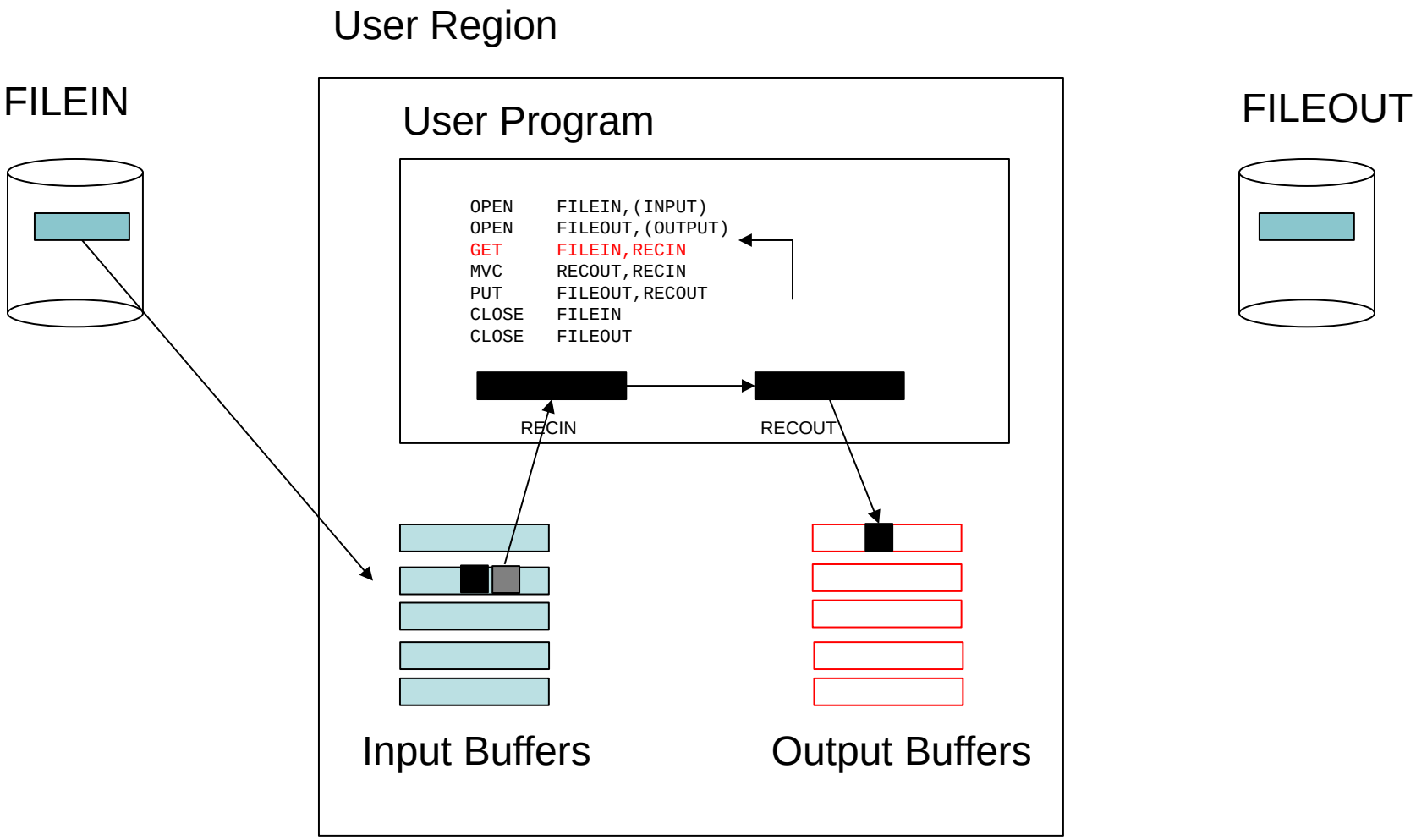
QSAM Move Mode I/O



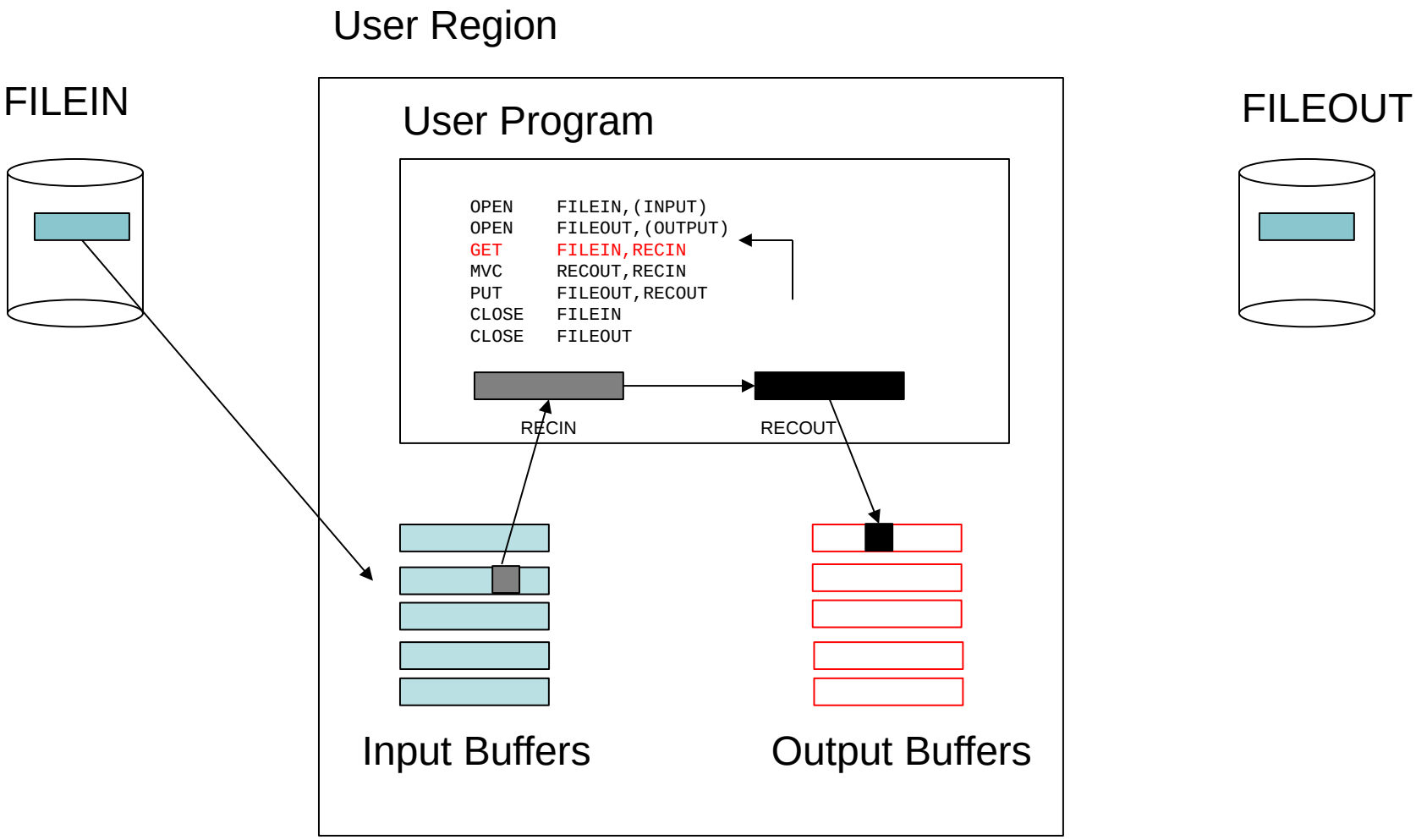
QSAM Move Mode I/O



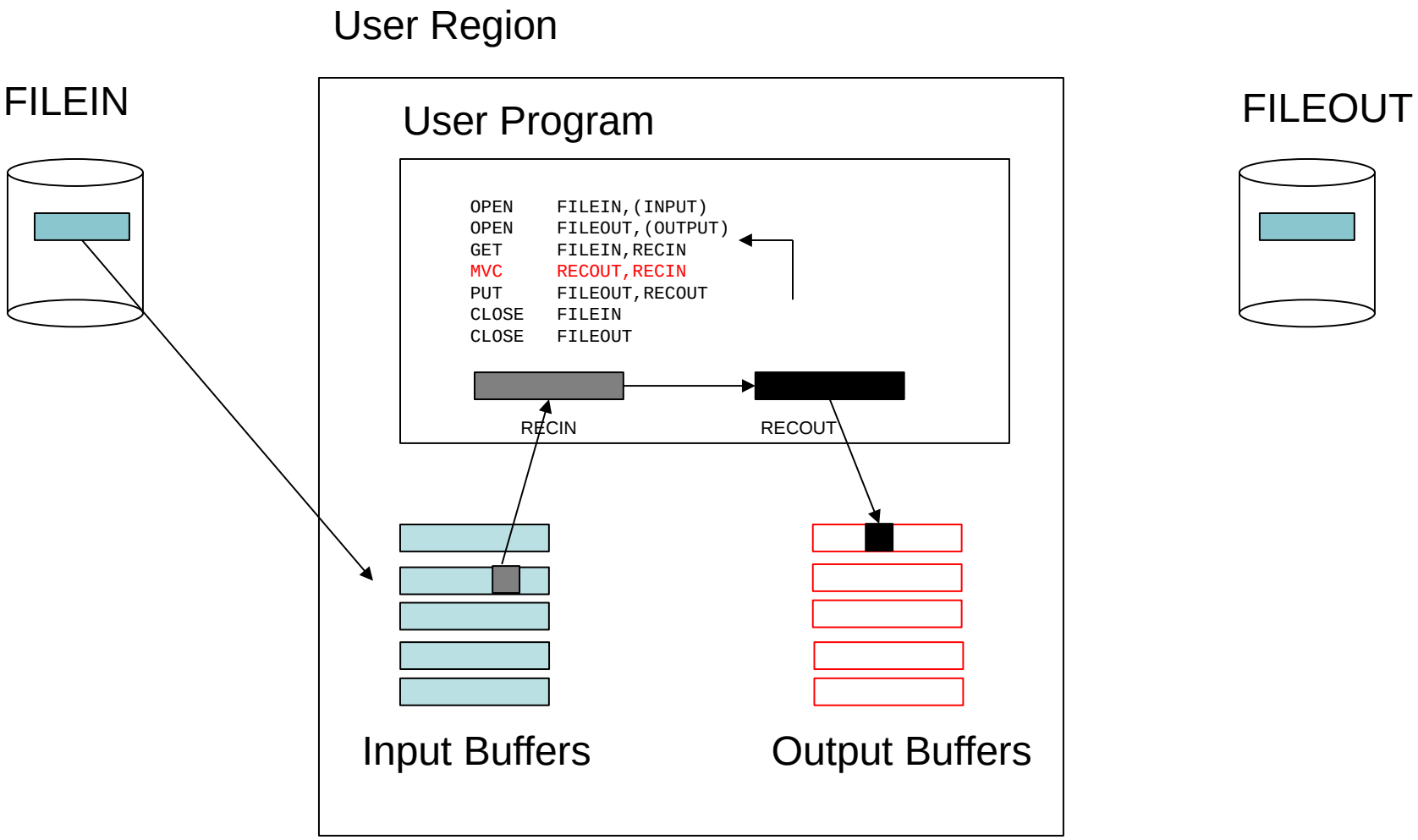
QSAM Move Mode I/O



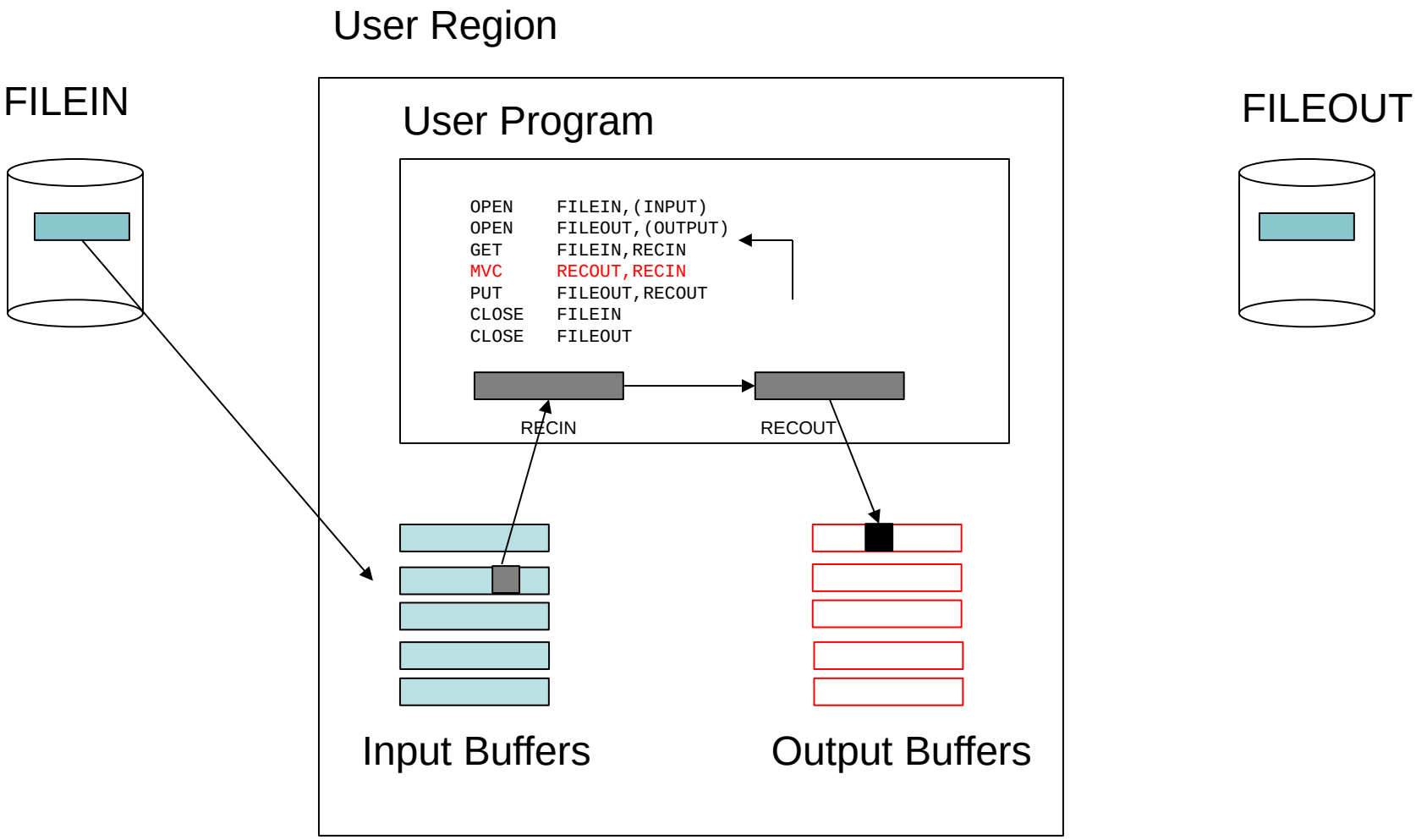
QSAM Move Mode I/O



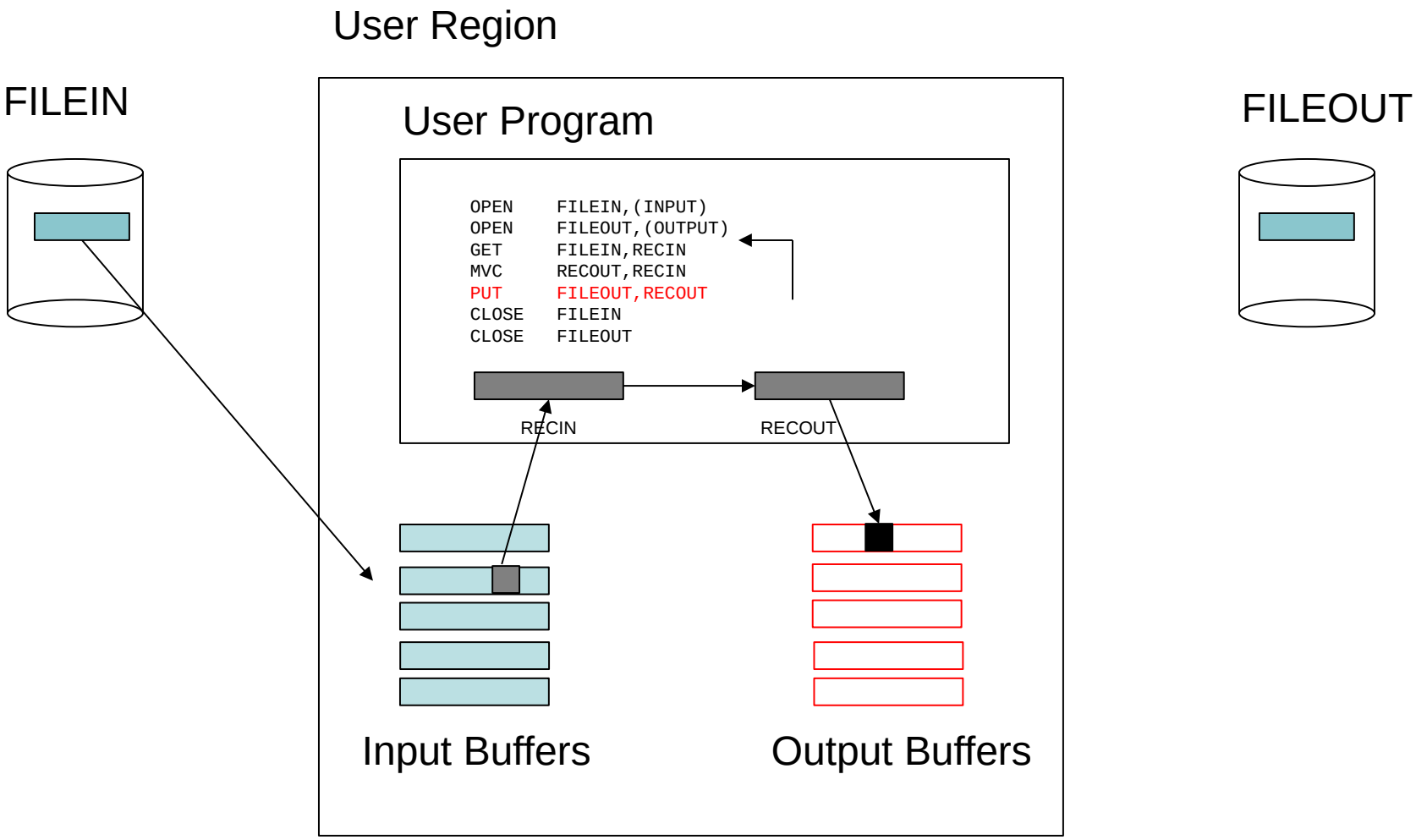
QSAM Move Mode I/O



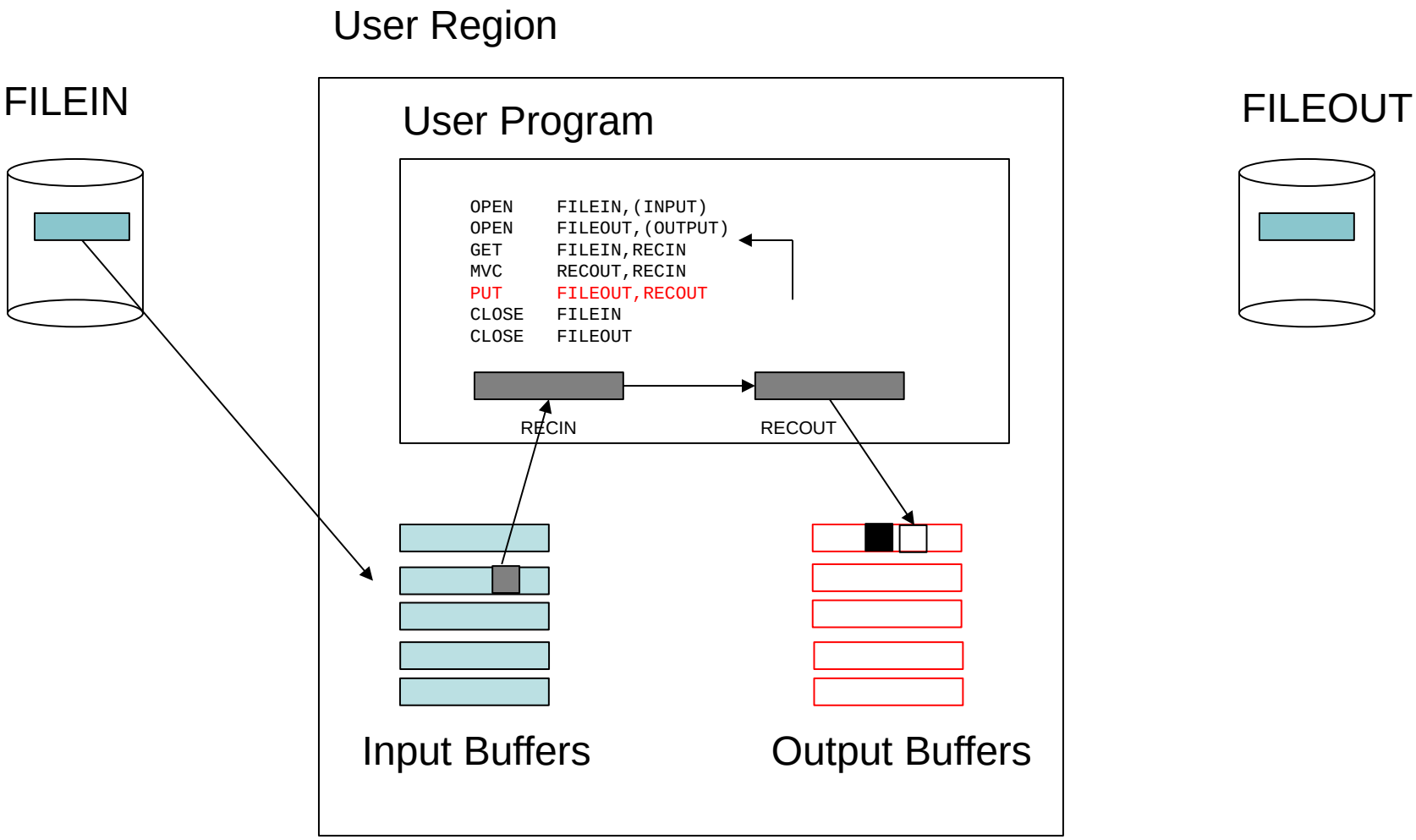
QSAM Move Mode I/O



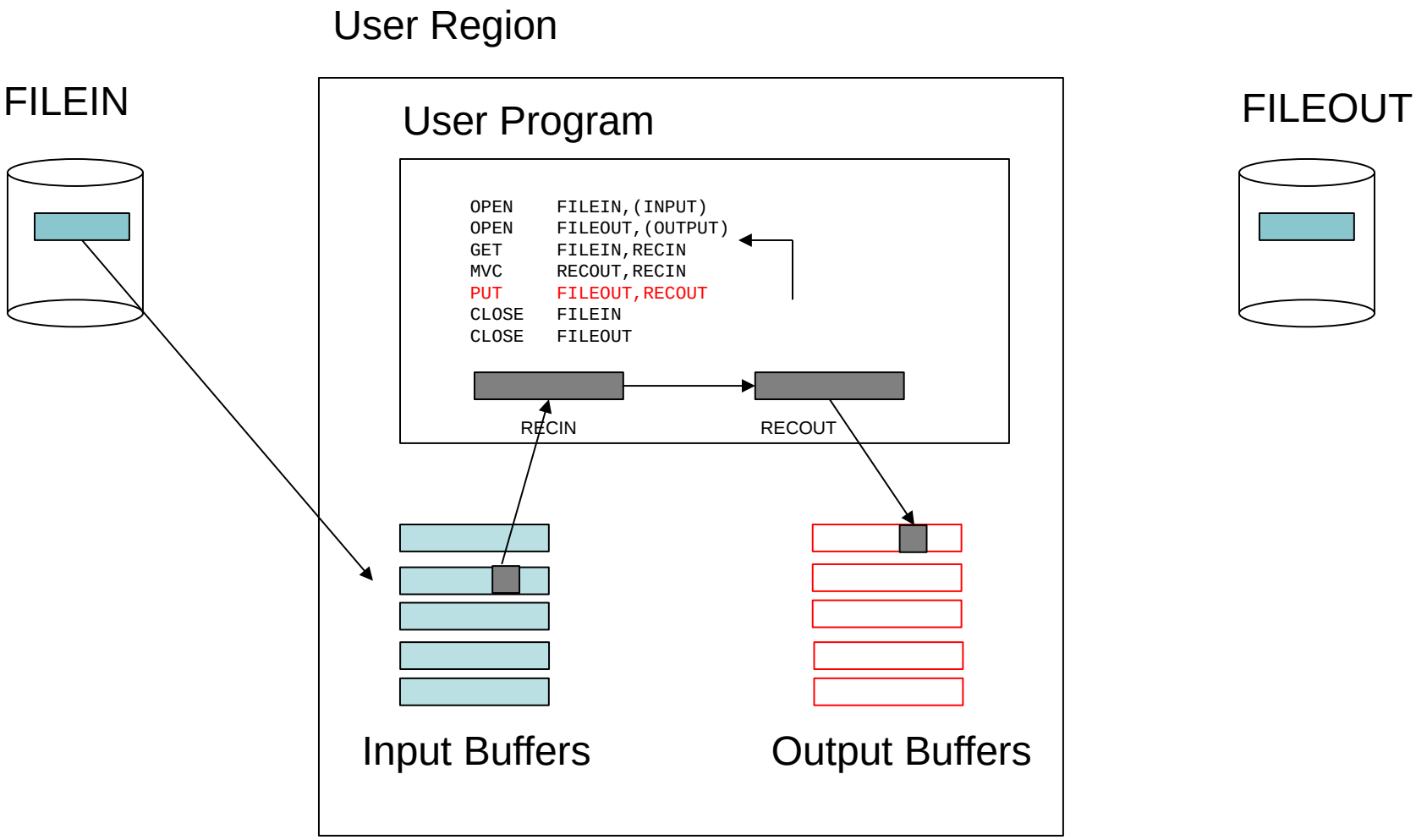
QSAM Move Mode I/O



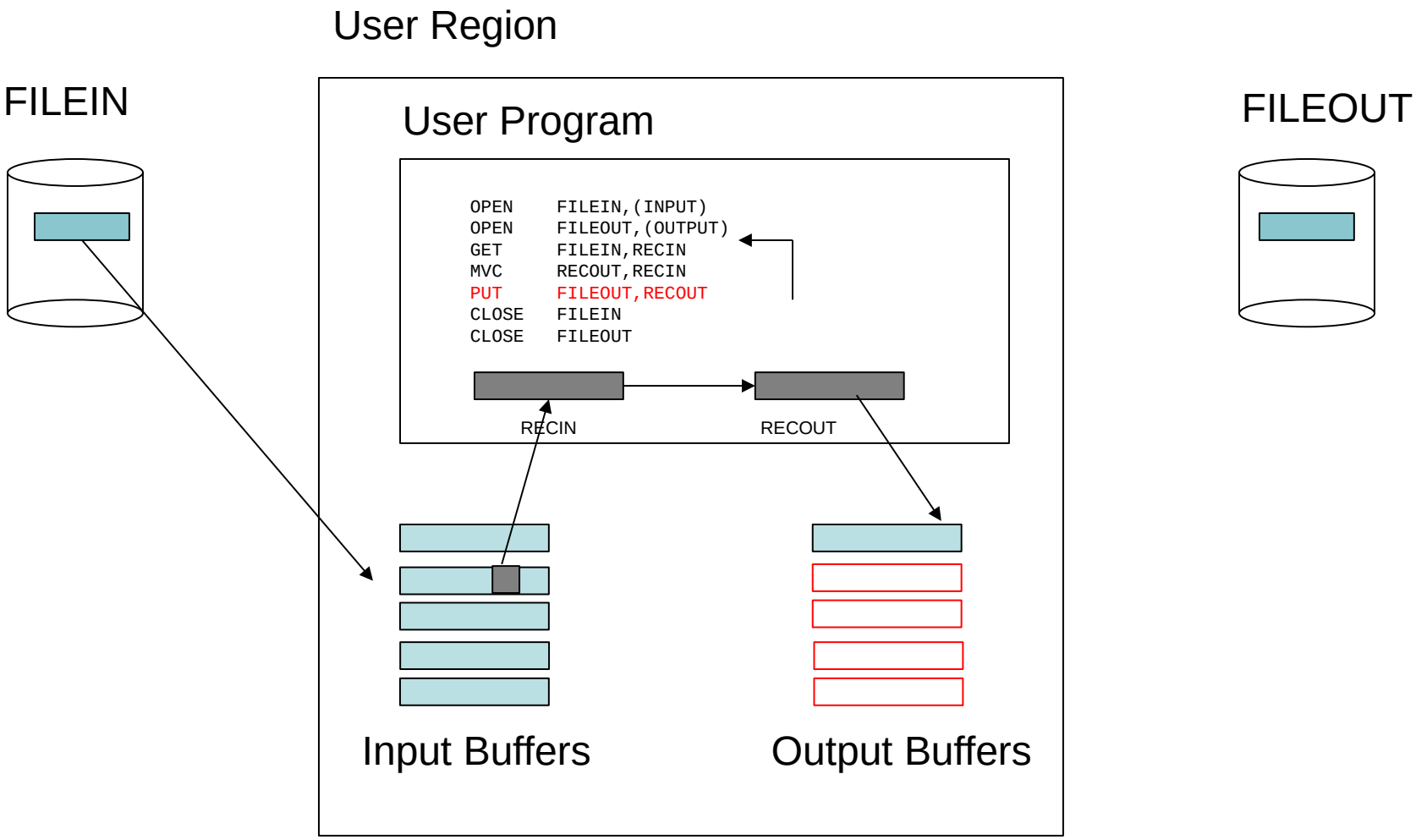
QSAM Move Mode I/O



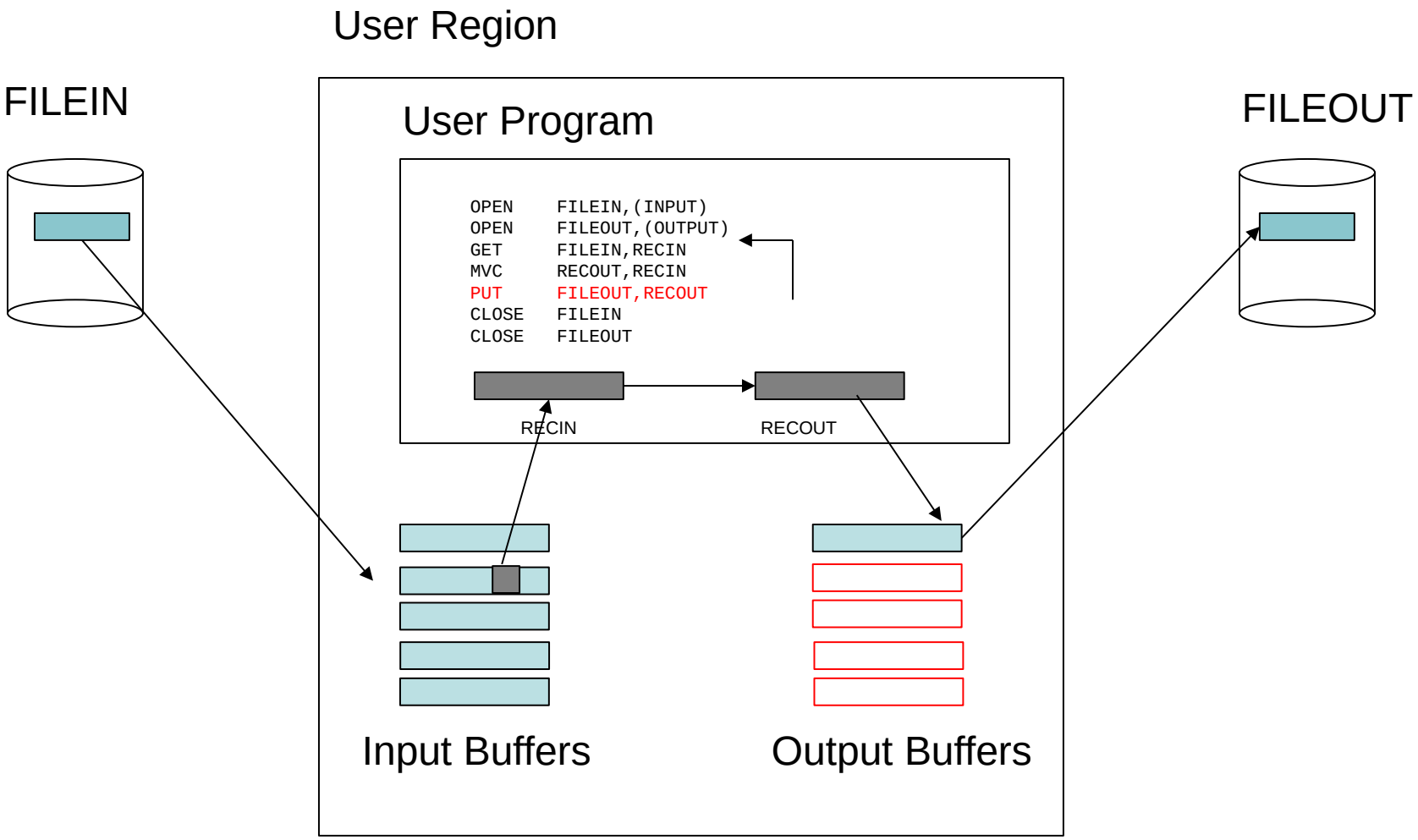
QSAM Move Mode I/O



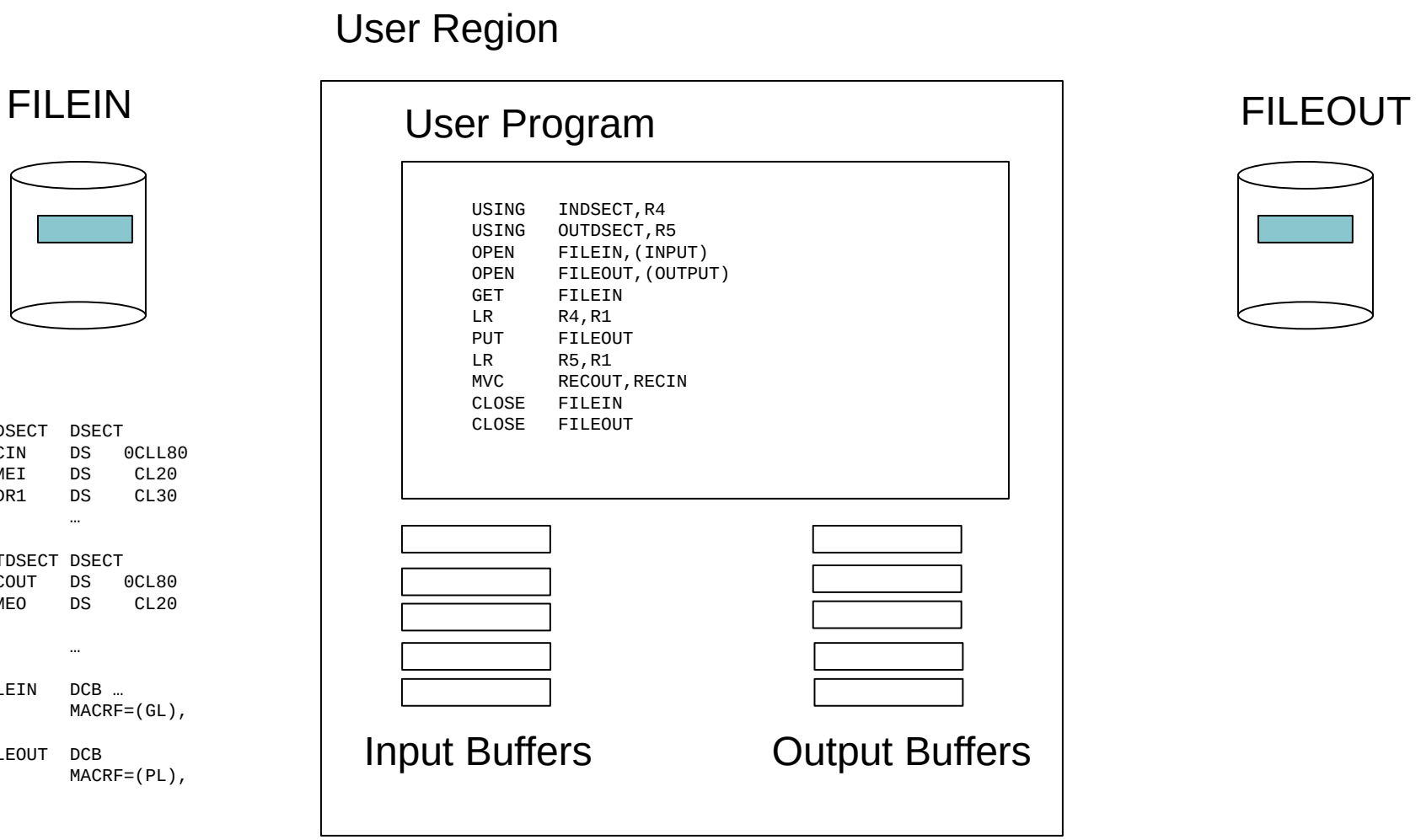
QSAM Move Mode I/O



QSAM Move Mode I/O



QSAM Locate Mode I/O

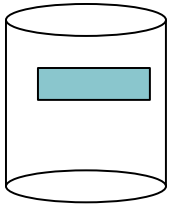



```
INDSECT  DSECT
RECIN    DS   0CLL80
NAMEI    DS    CL20
ADDR1    DS    CL30
         …

OUTDSECT DSECT
RECOUT   DS   0CL80
NAMEO    DS    CL20

         …

FILEIN   DCB …
         MACRF=(GL),

FILEOUT  DCB
         MACRF=(PL),
```

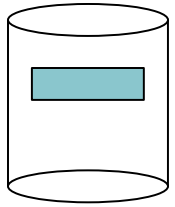
QSAM Locate Mode I/O

User Region

FILEIN



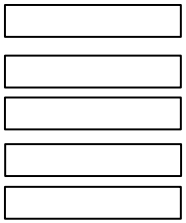
FILEOUT



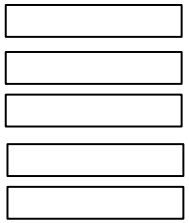
```
INDSECT  DSECT
RECIN    DS   0CL80
NAMEI    DS    CL20
ADDR1    DS    CL30
         …

OUTDSECT DSECT
RECOUT   DS   0CL80
NAMEO    DS    CL20

         …

FILEIN   DCB …
         MACRF=(GL),

FILEOUT  DCB
         MACRF=(PL),
```

User Program

```
USING   INDSECT,R4
USING   OUTDSECT,R5
OPEN    FILEIN,(INPUT)
OPEN    FILEOUT,(OUTPUT)
GET     FILEIN
LR      R4,R1
PUT     FILEOUT
LR      R5,R1
MVC     RECOUT,RECIN
CLOSE   FILEIN
CLOSE   FILEOUT
```

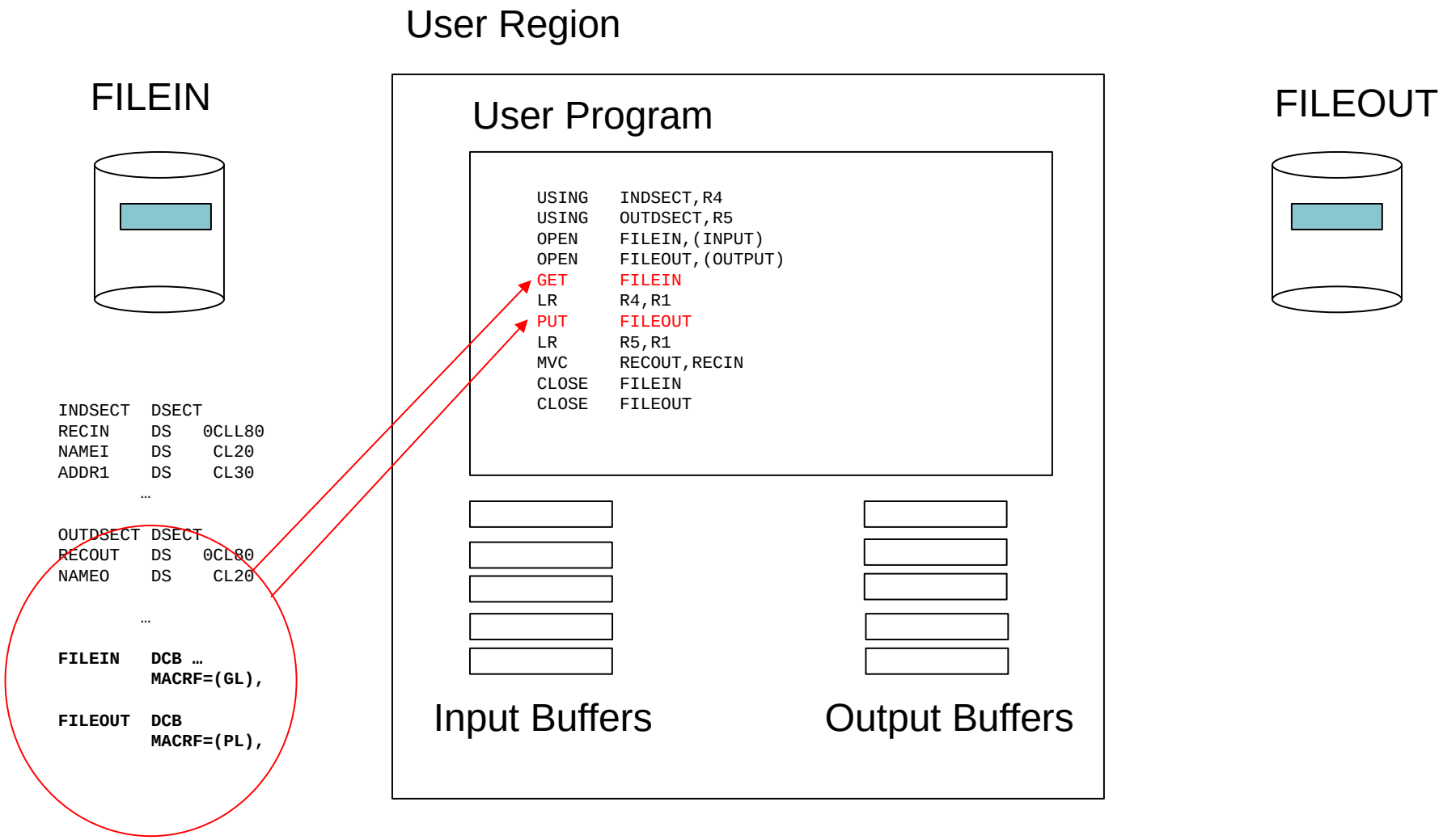


Input Buffers

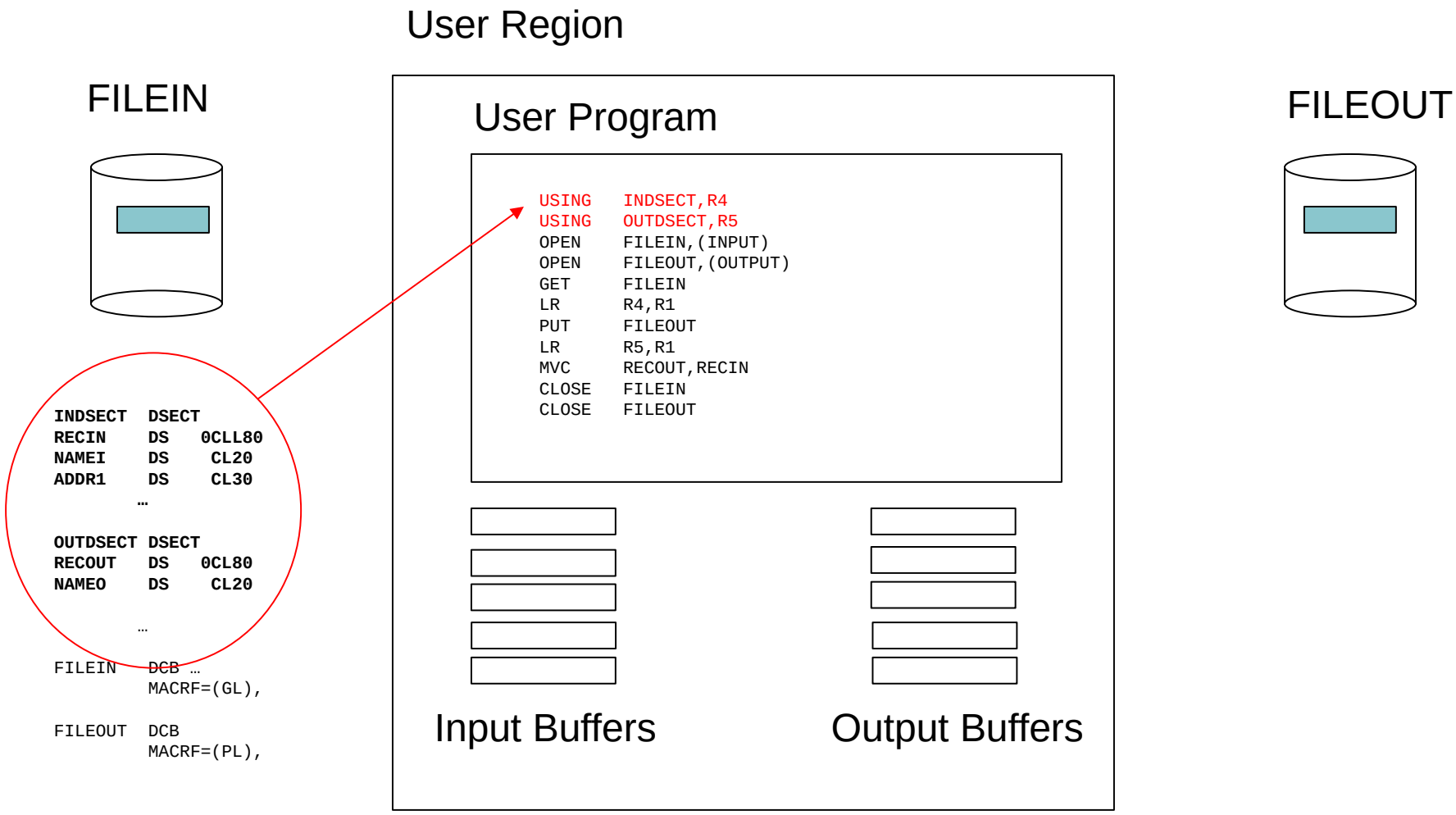


Output Buffers

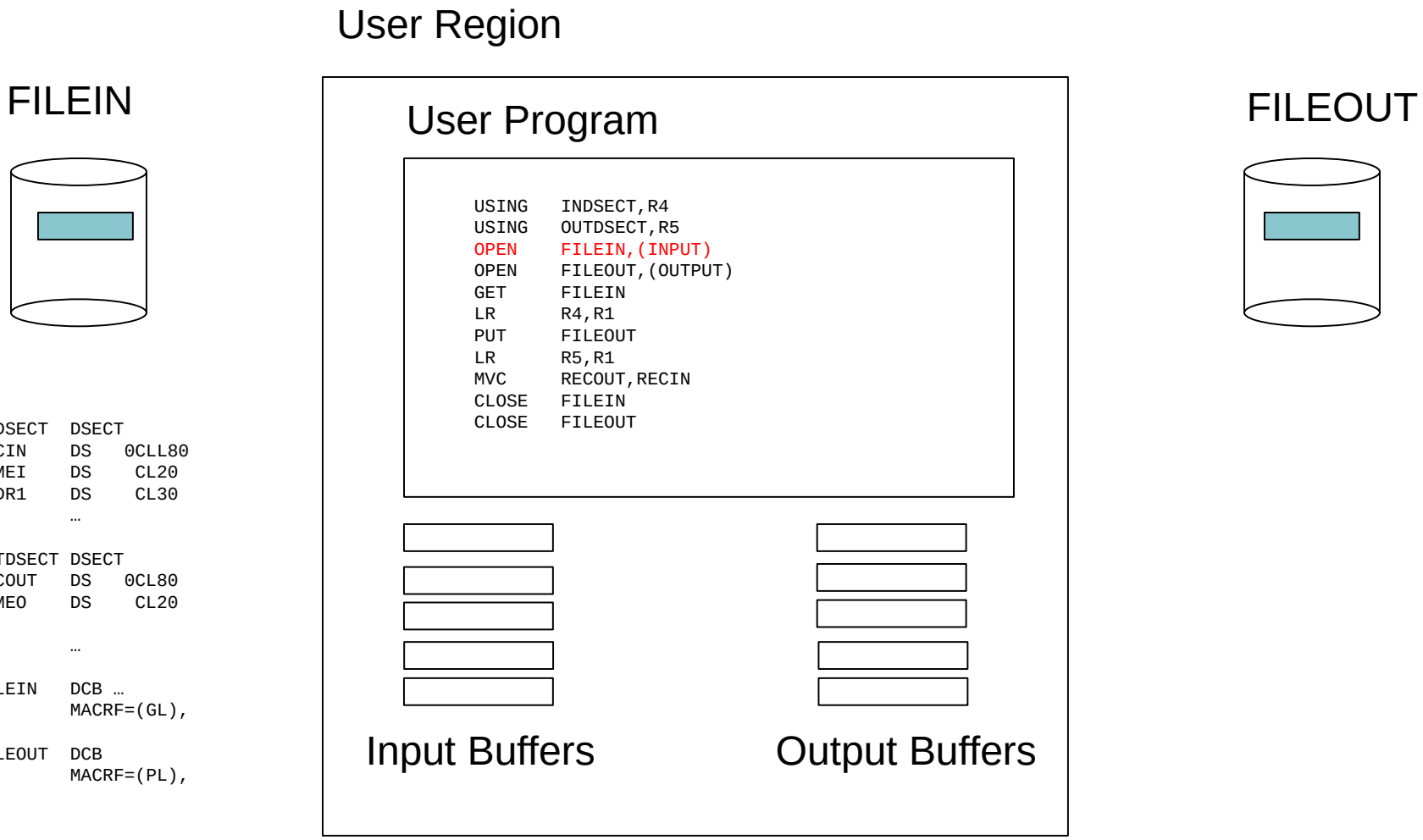
QSAM Locate Mode I/O



QSAM Locate Mode I/O

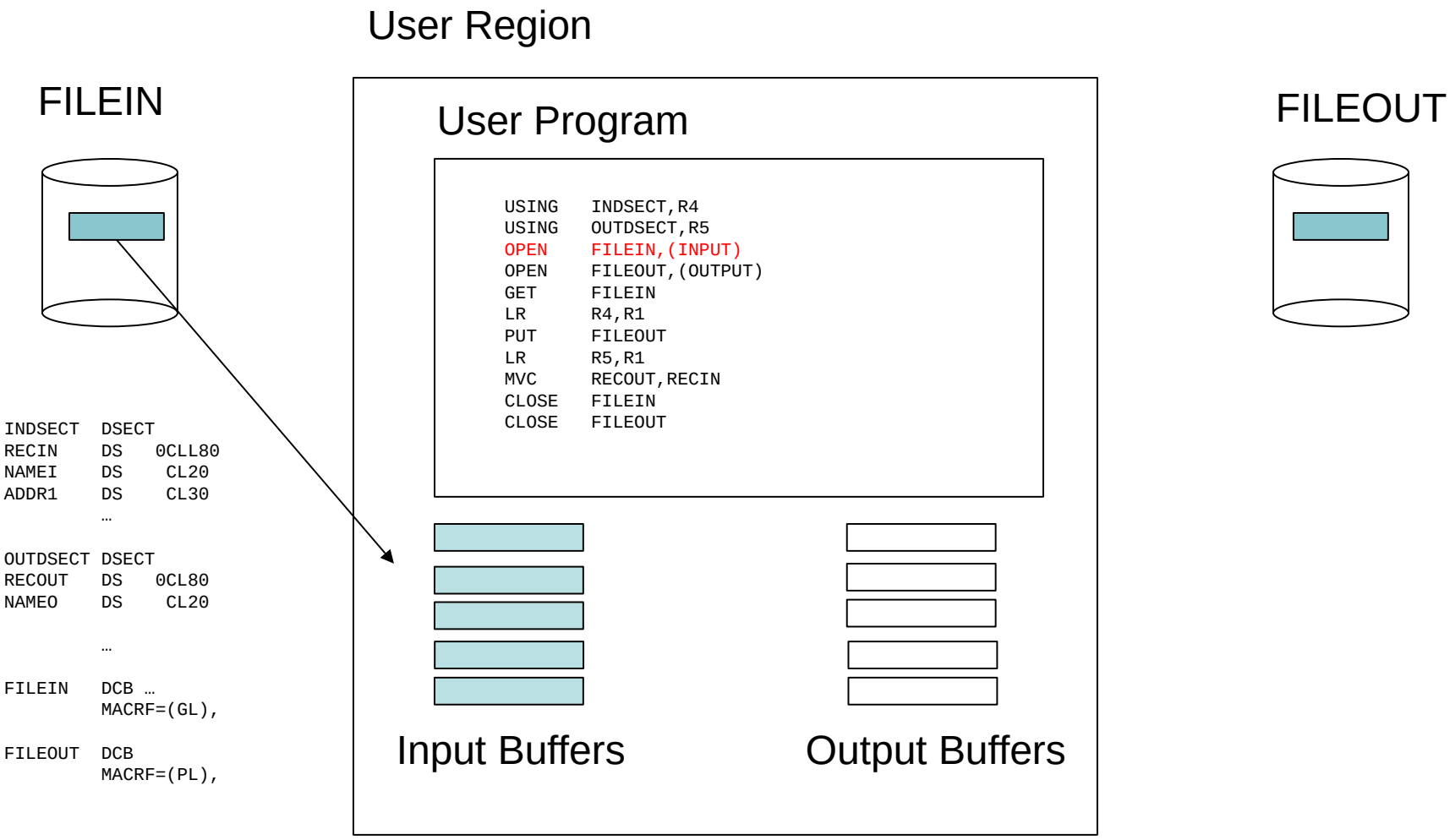


QSAM Locate Mode I/O

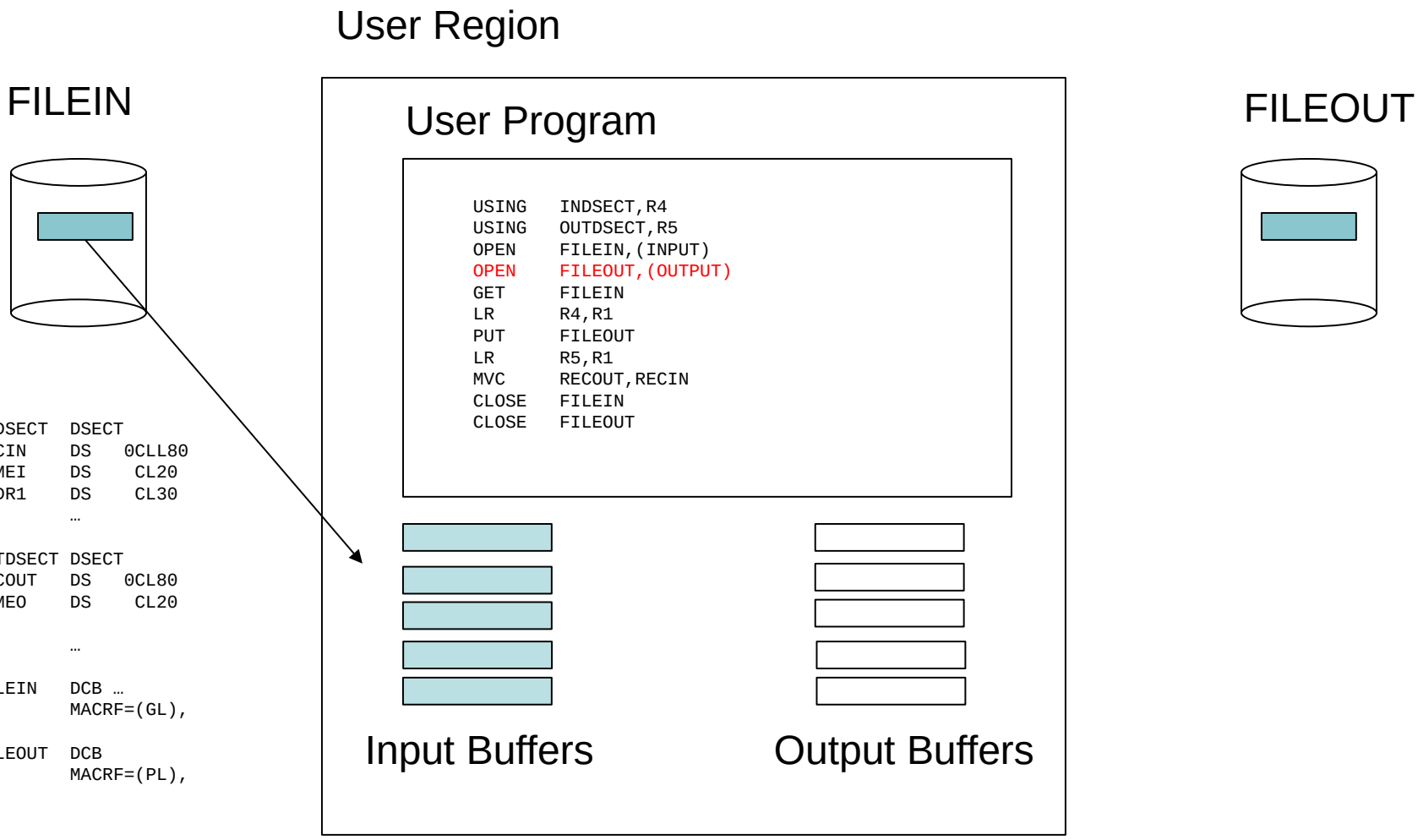



```
INDSECT  DSECT
RECIN    DS   0CLL80
NAMEI    DS    CL20
ADDR1    DS    CL30
         …

OUTDSECT DSECT
RECOUT   DS   0CL80
NAMEO    DS    CL20

         …

FILEIN   DCB …
         MACRF=(GL),

FILEOUT  DCB
         MACRF=(PL),
```

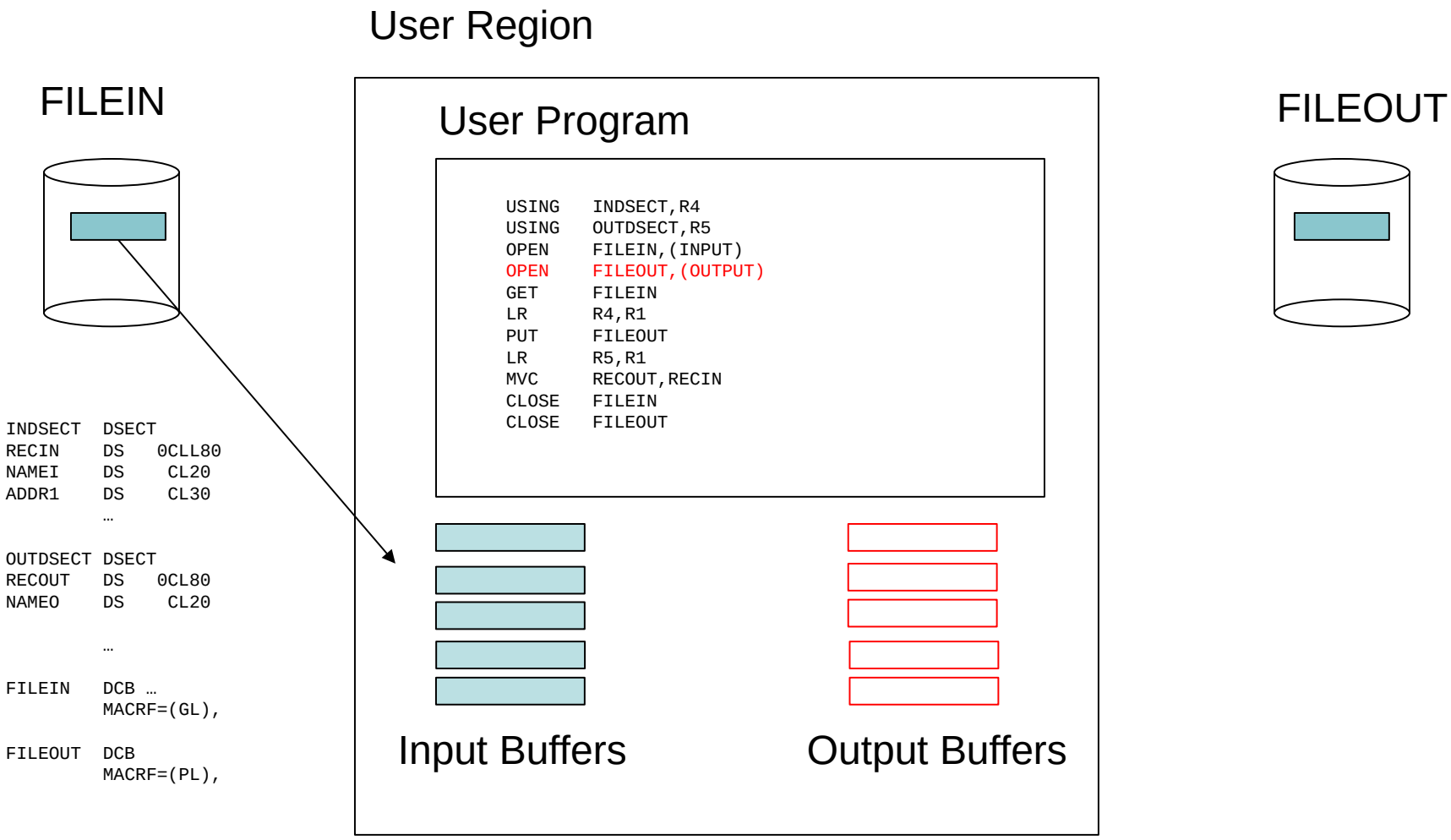
QSAM Locate Mode I/O



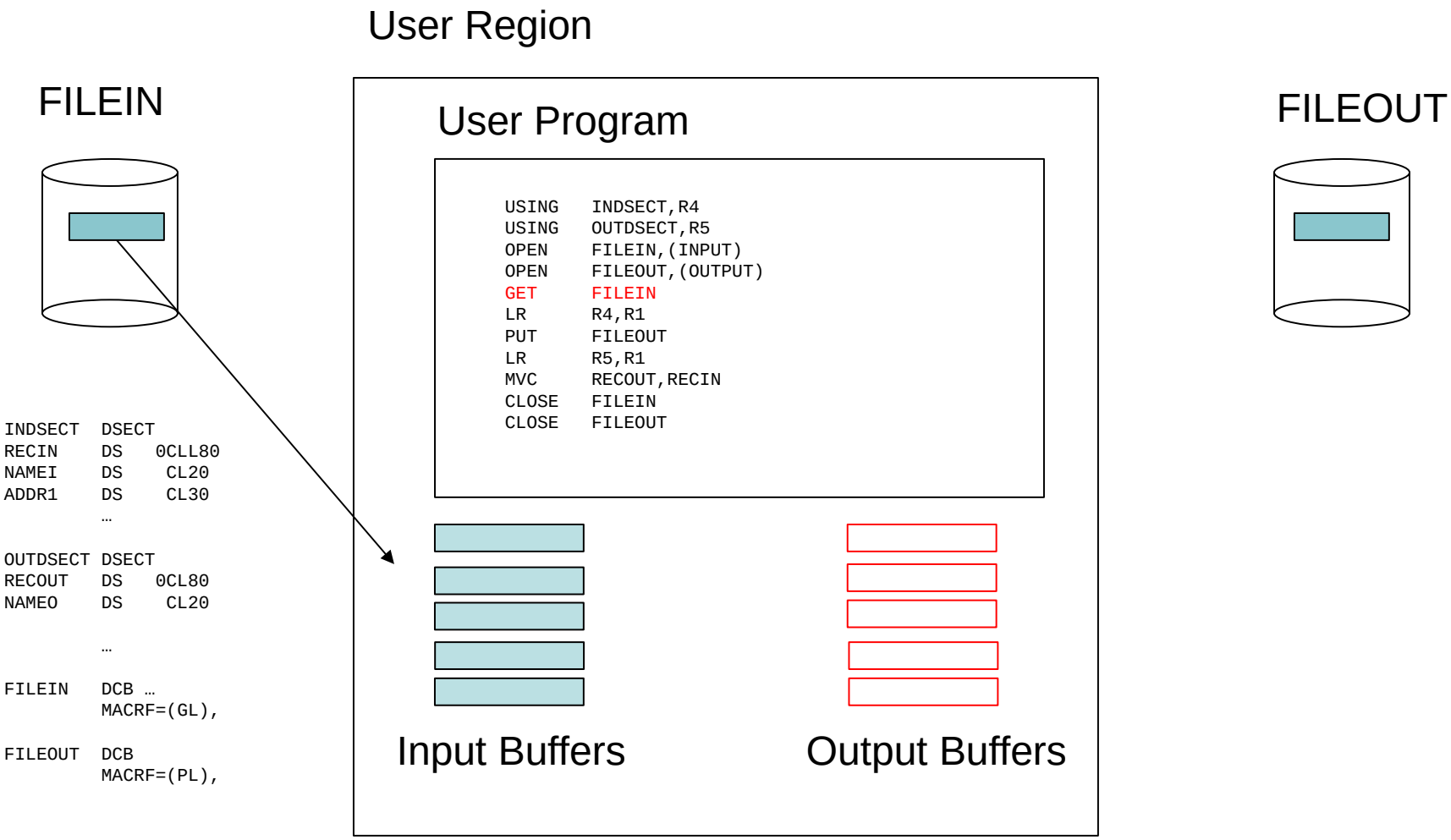
QSAM Locate Mode I/O



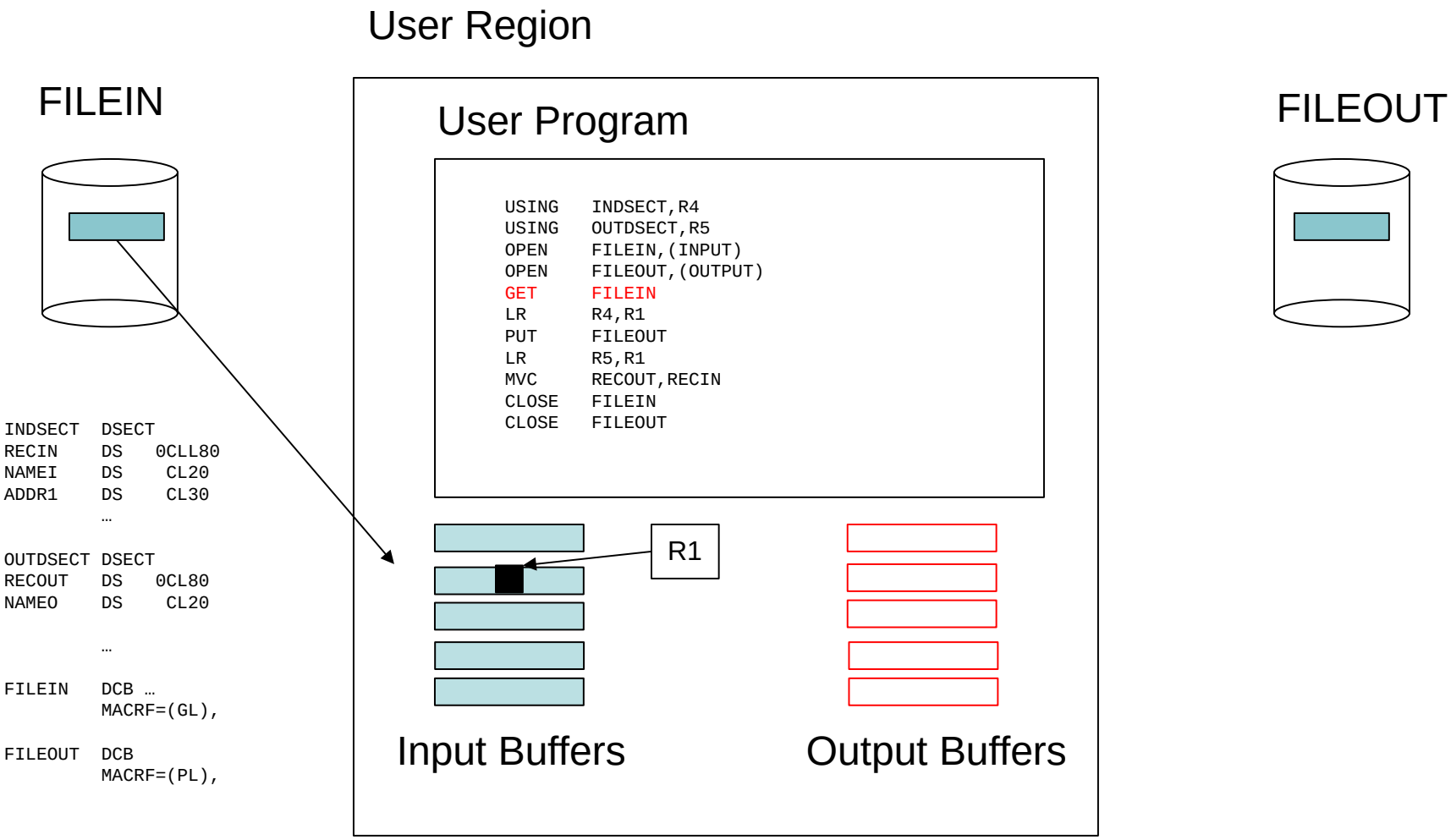
QSAM Locate Mode I/O



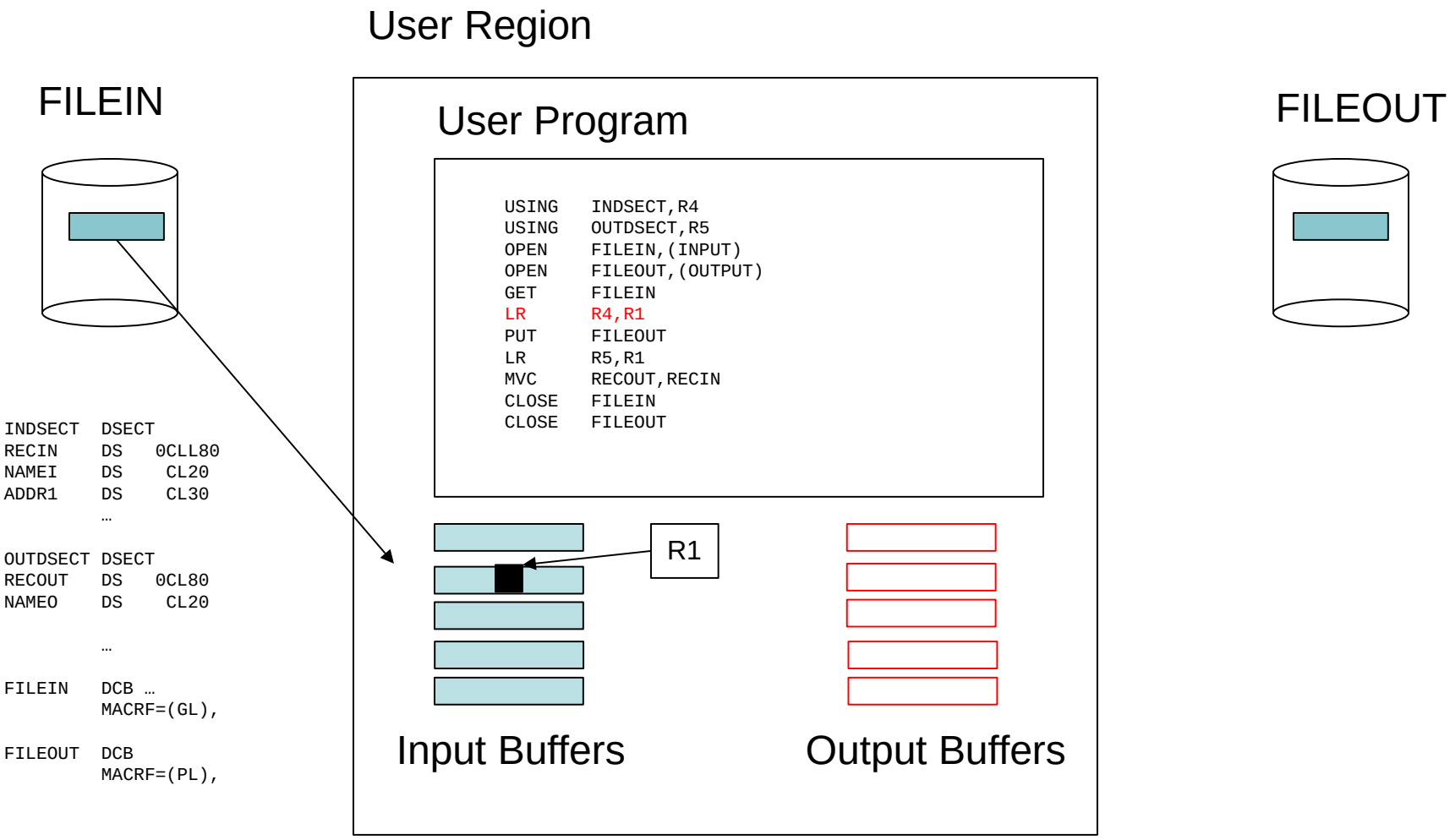
QSAM Locate Mode I/O



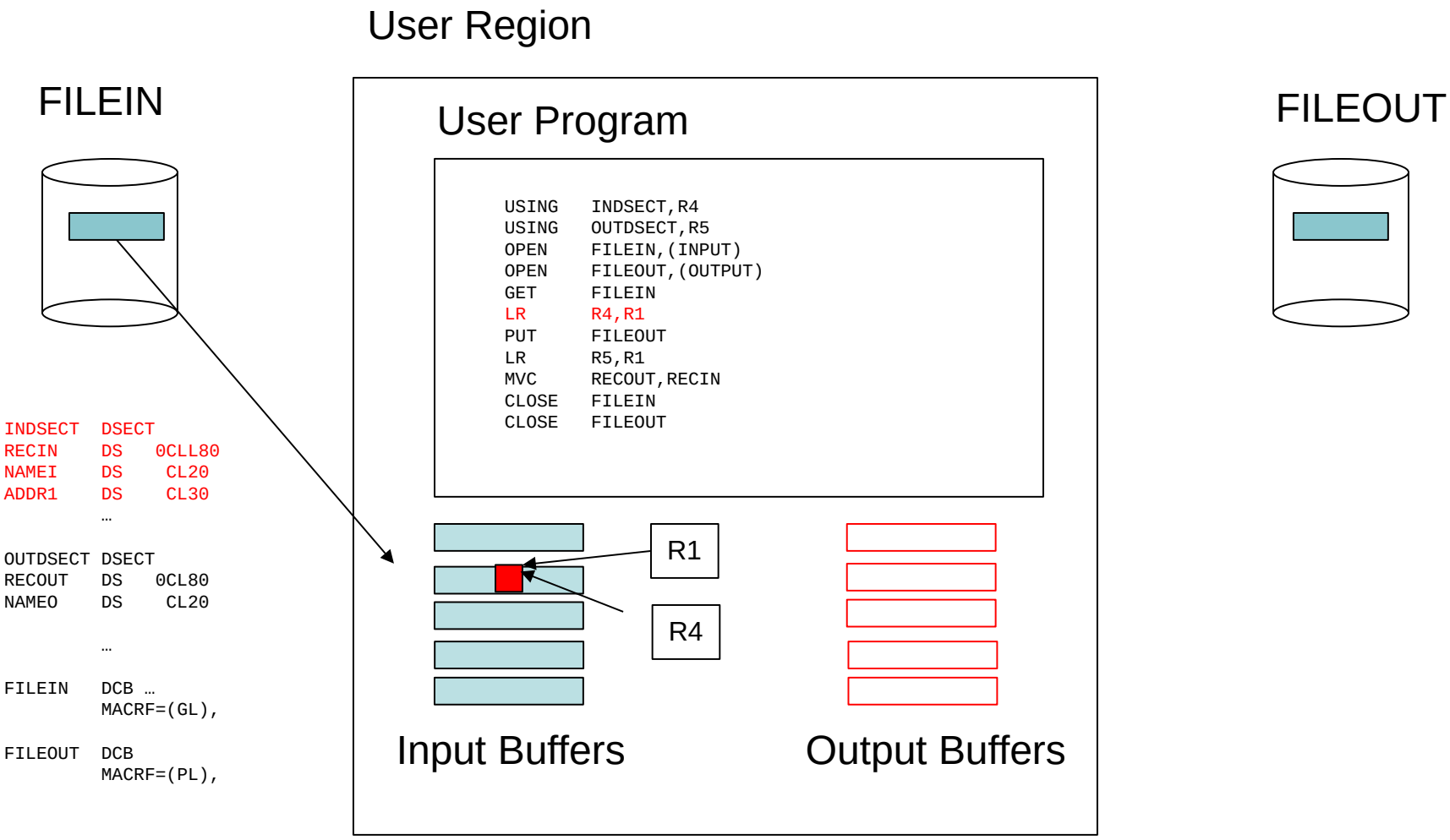
QSAM Locate Mode I/O



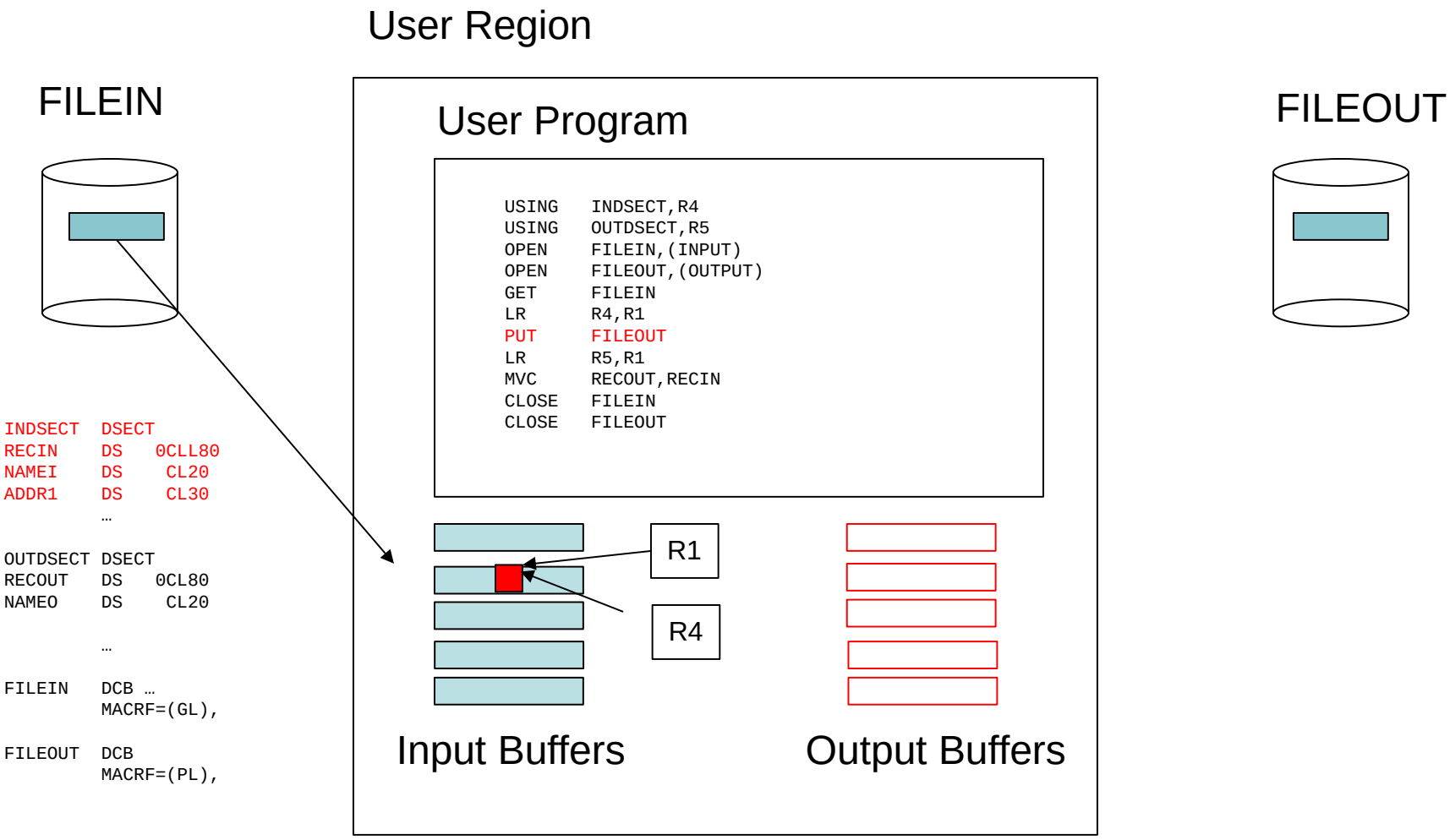
QSAM Locate Mode I/O



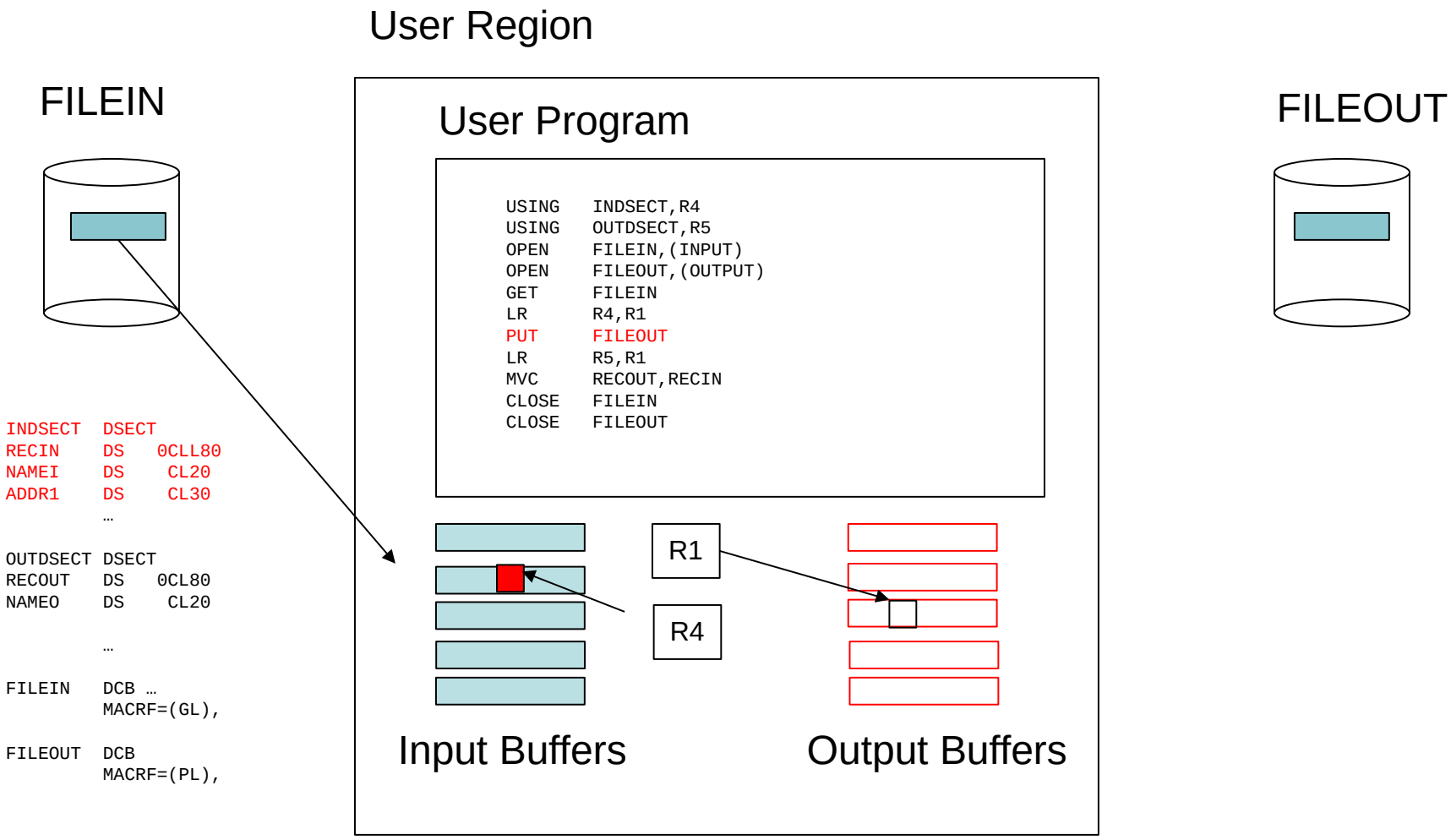
QSAM Locate Mode I/O



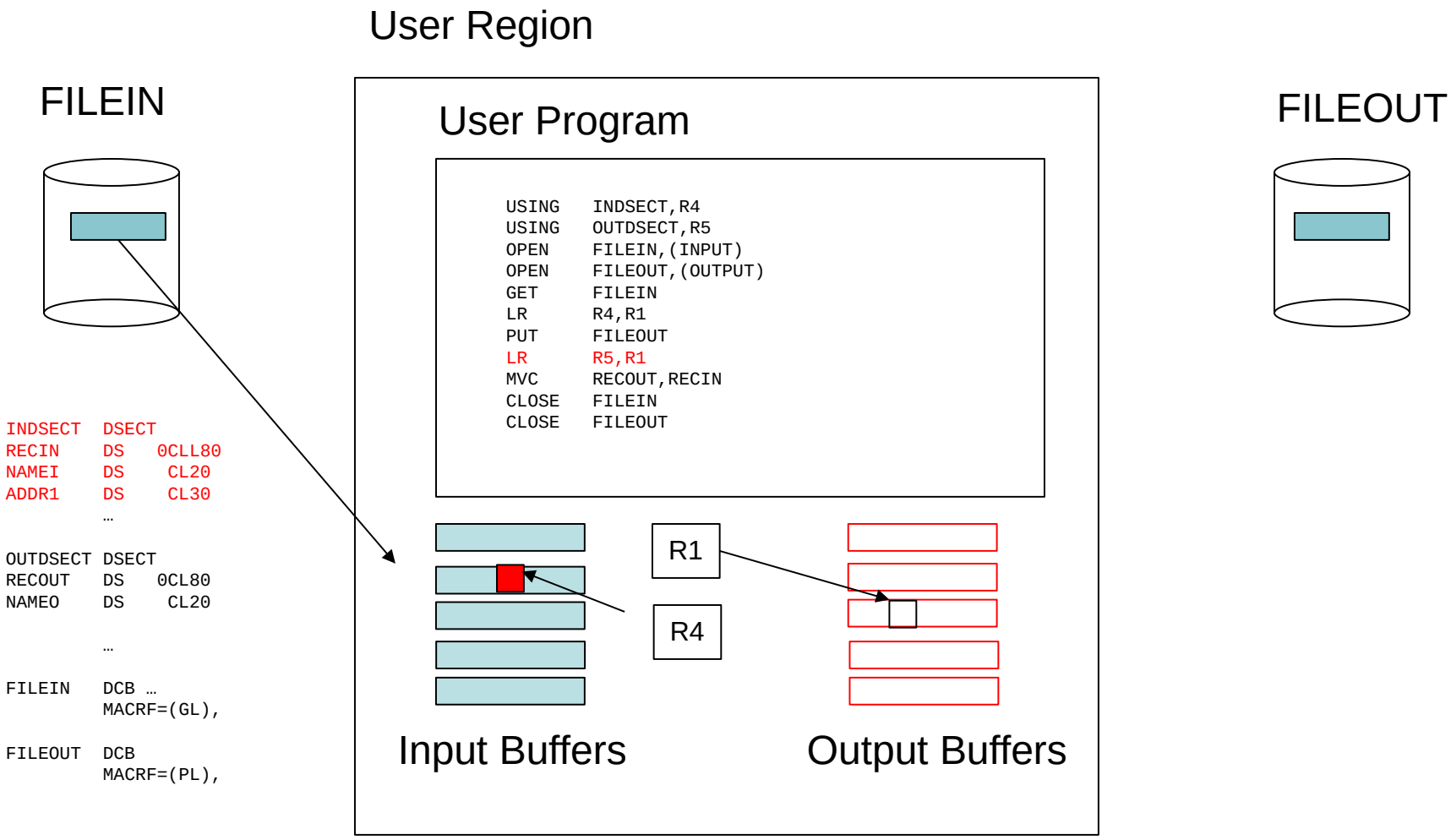
QSAM Locate Mode I/O



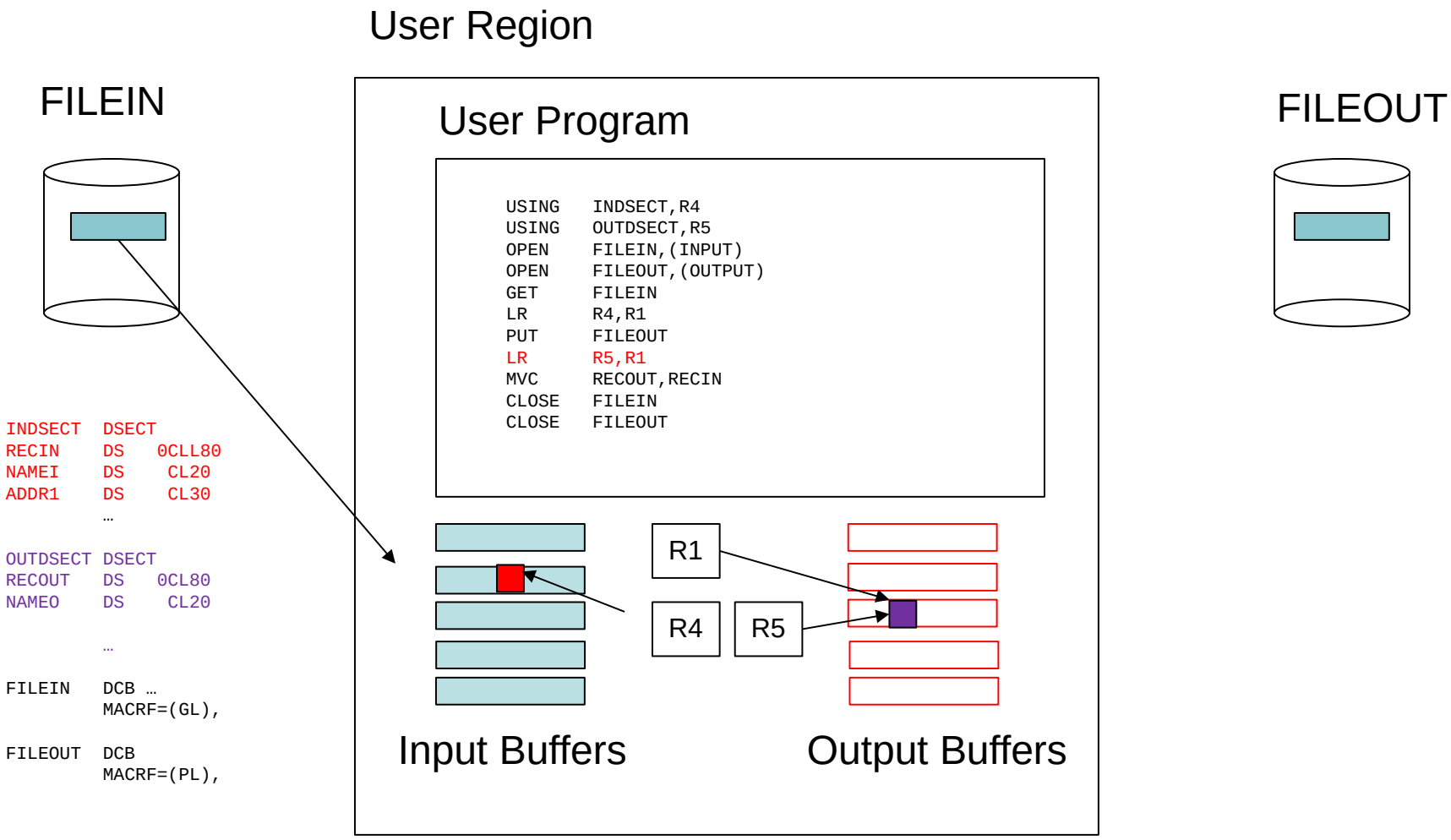
QSAM Locate Mode I/O



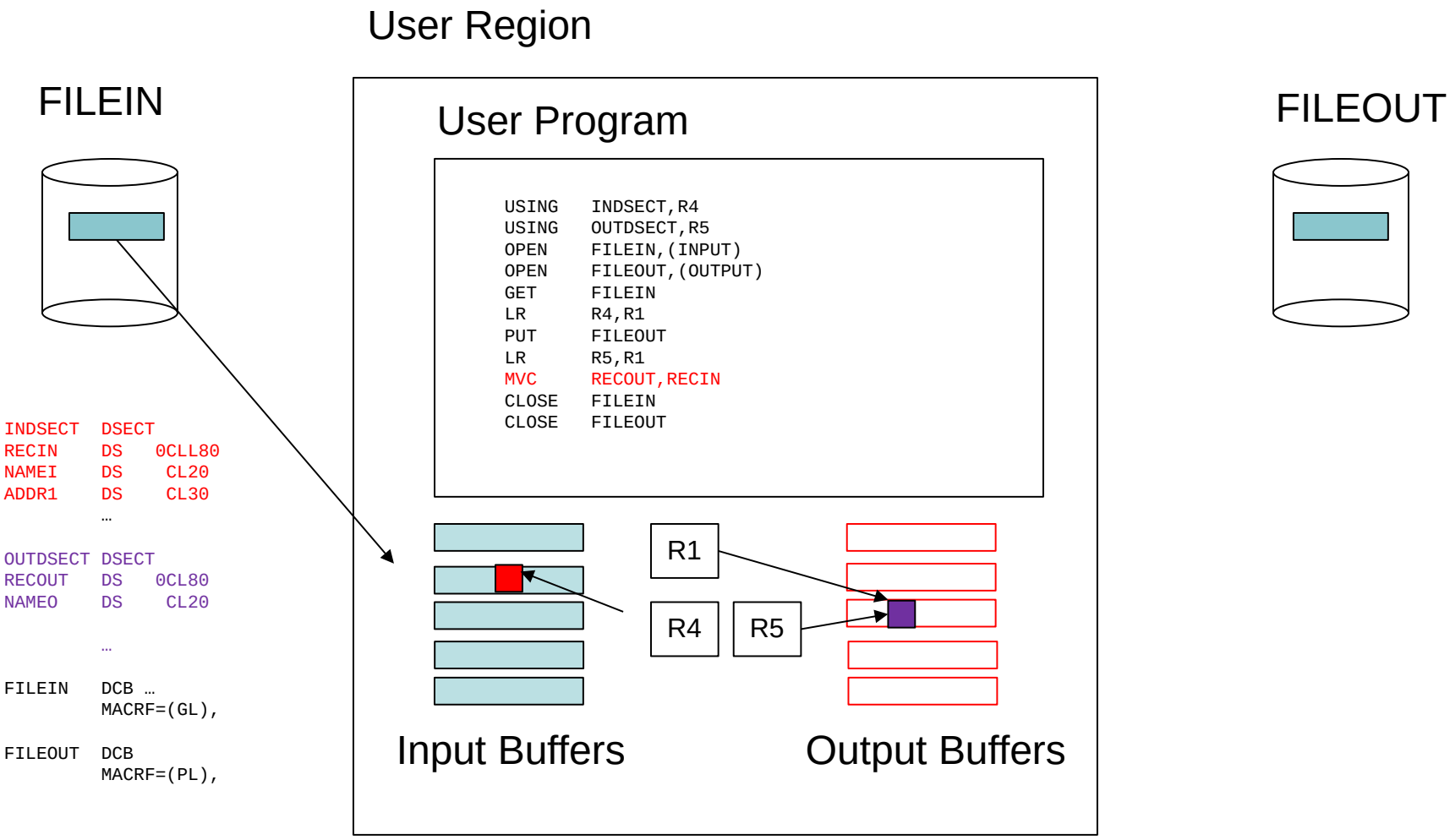
QSAM Locate Mode I/O



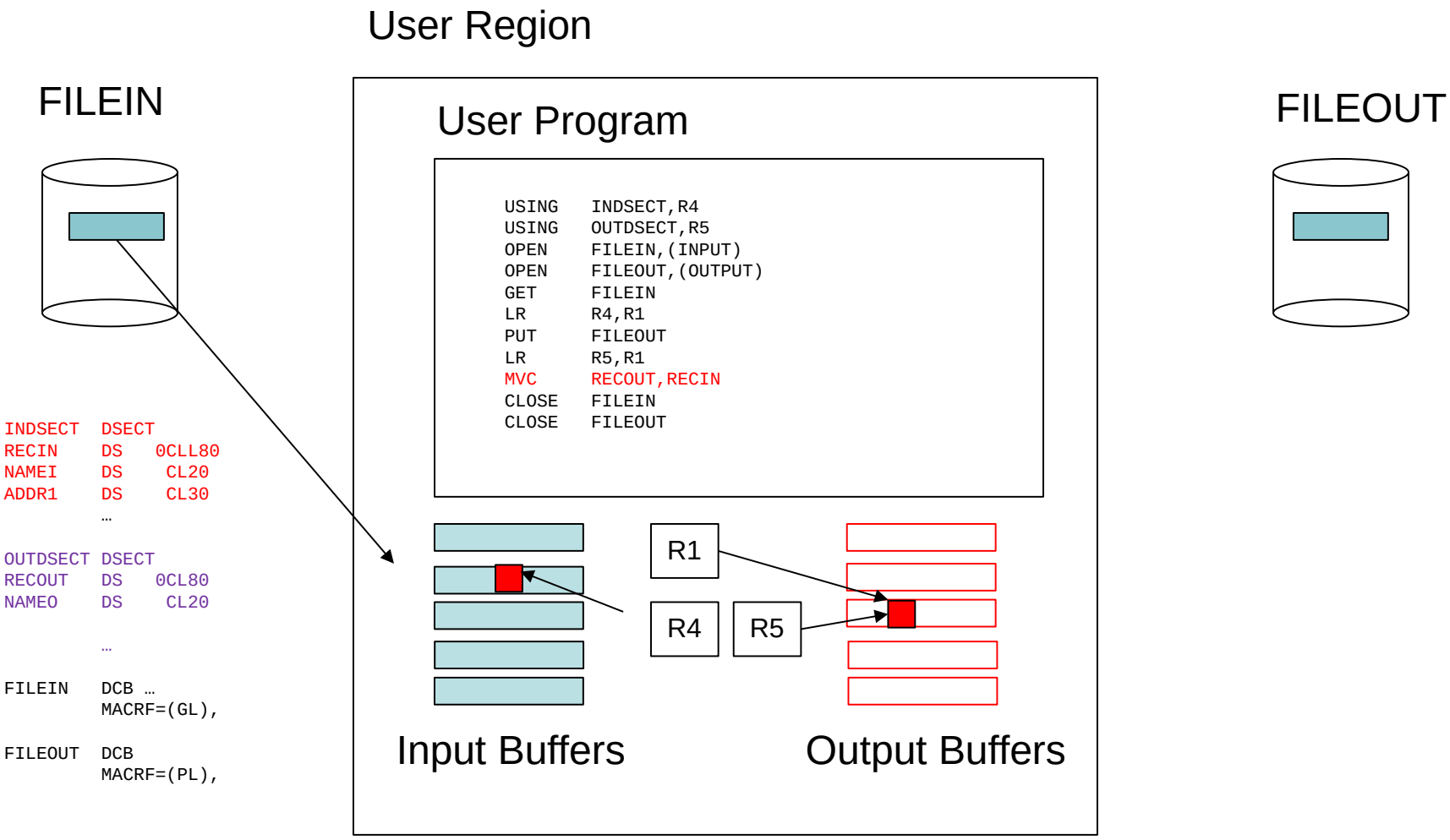
QSAM Locate Mode I/O



QSAM Locate Mode I/O

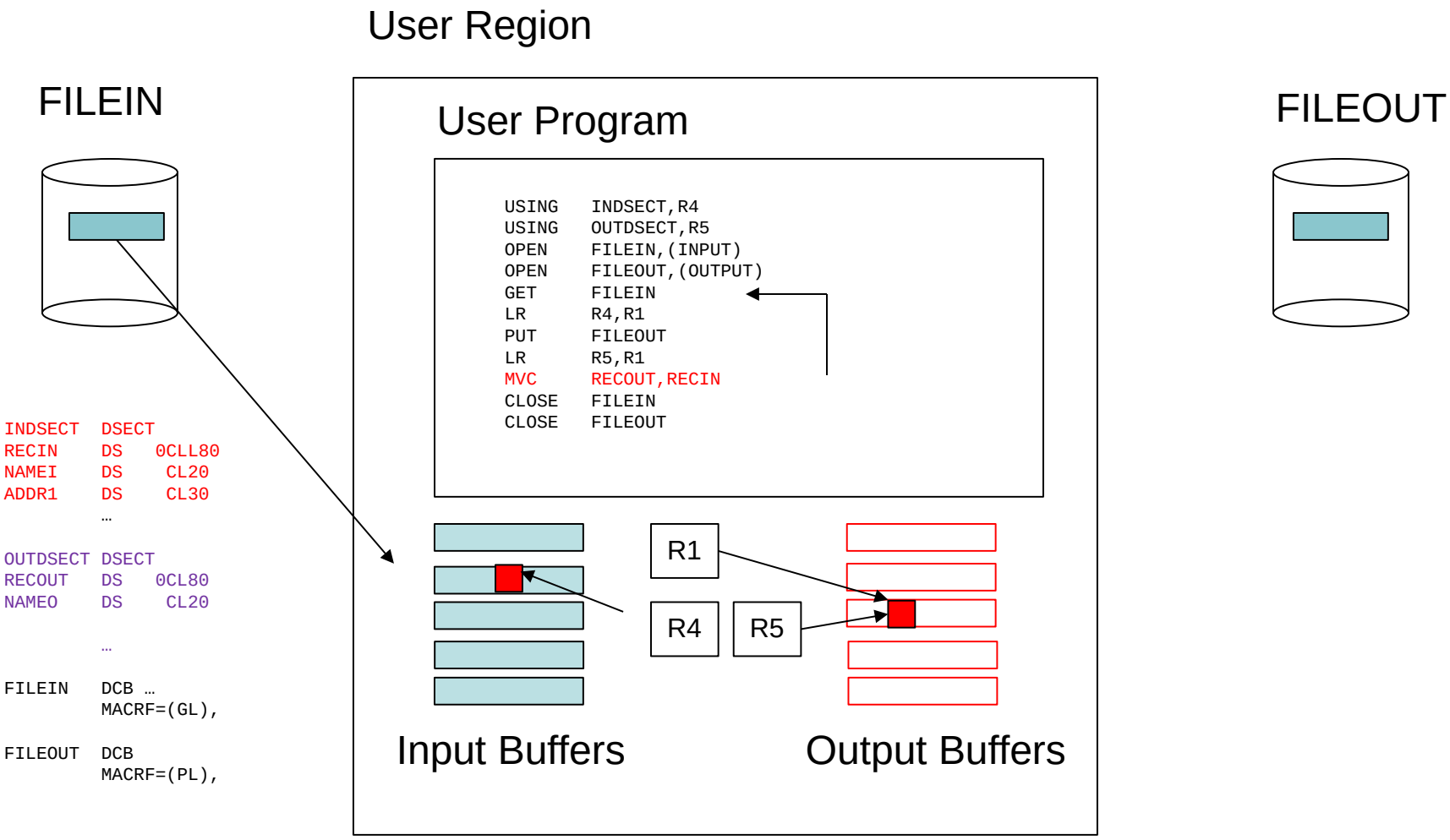


QSAM Locate Mode I/O

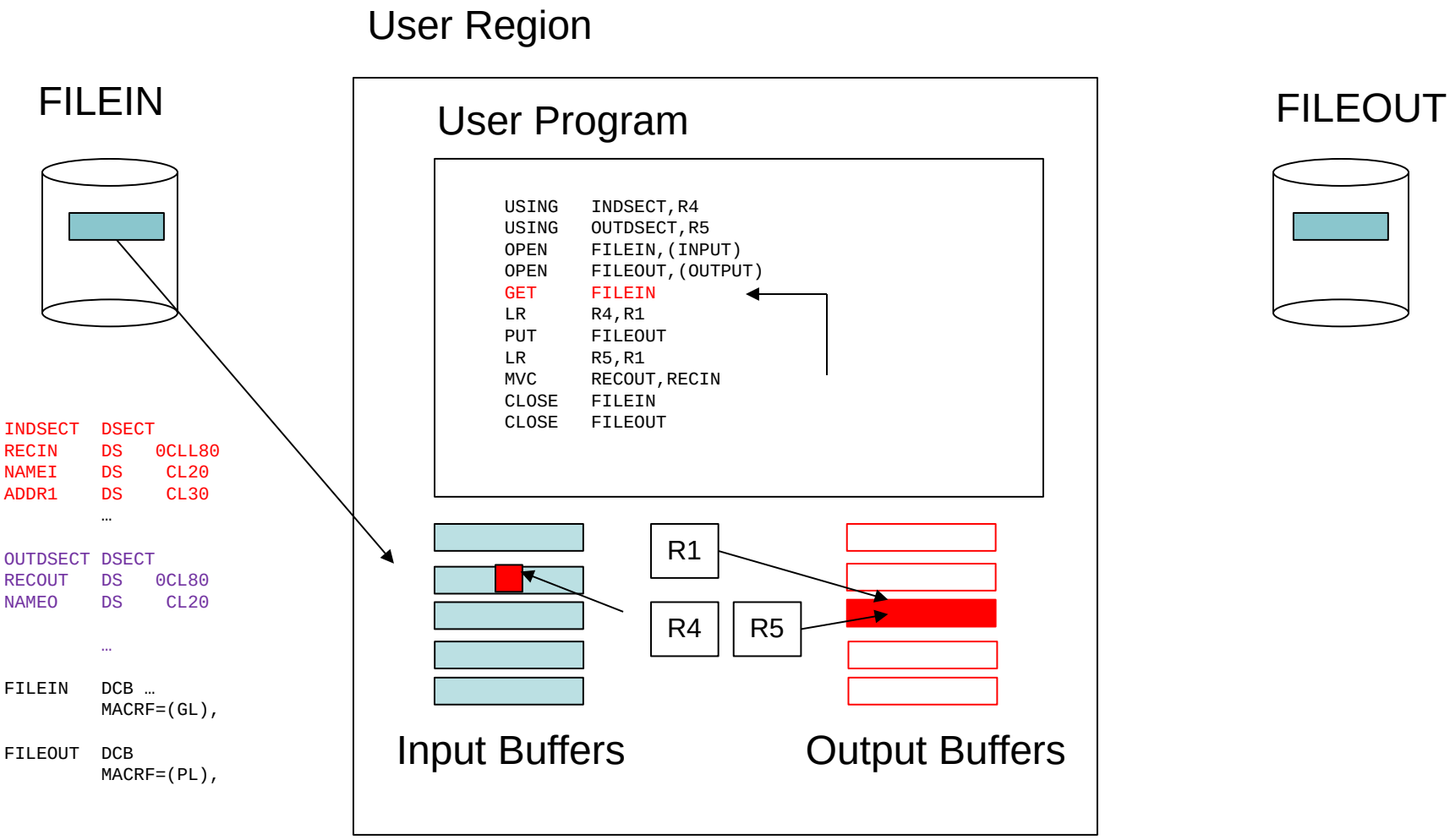



```
INDSECT  DSECT
RECIN    DS    0CLL80
NAMEI    DS     CL20
ADDR1    DS     CL30
         …

OUTDSECT DSECT
RECOUT   DS    0CL80
NAMEO    DS     CL20

         …

FILEIN   DCB …
         MACRF=(GL),

FILEOUT  DCB
         MACRF=(PL),
```

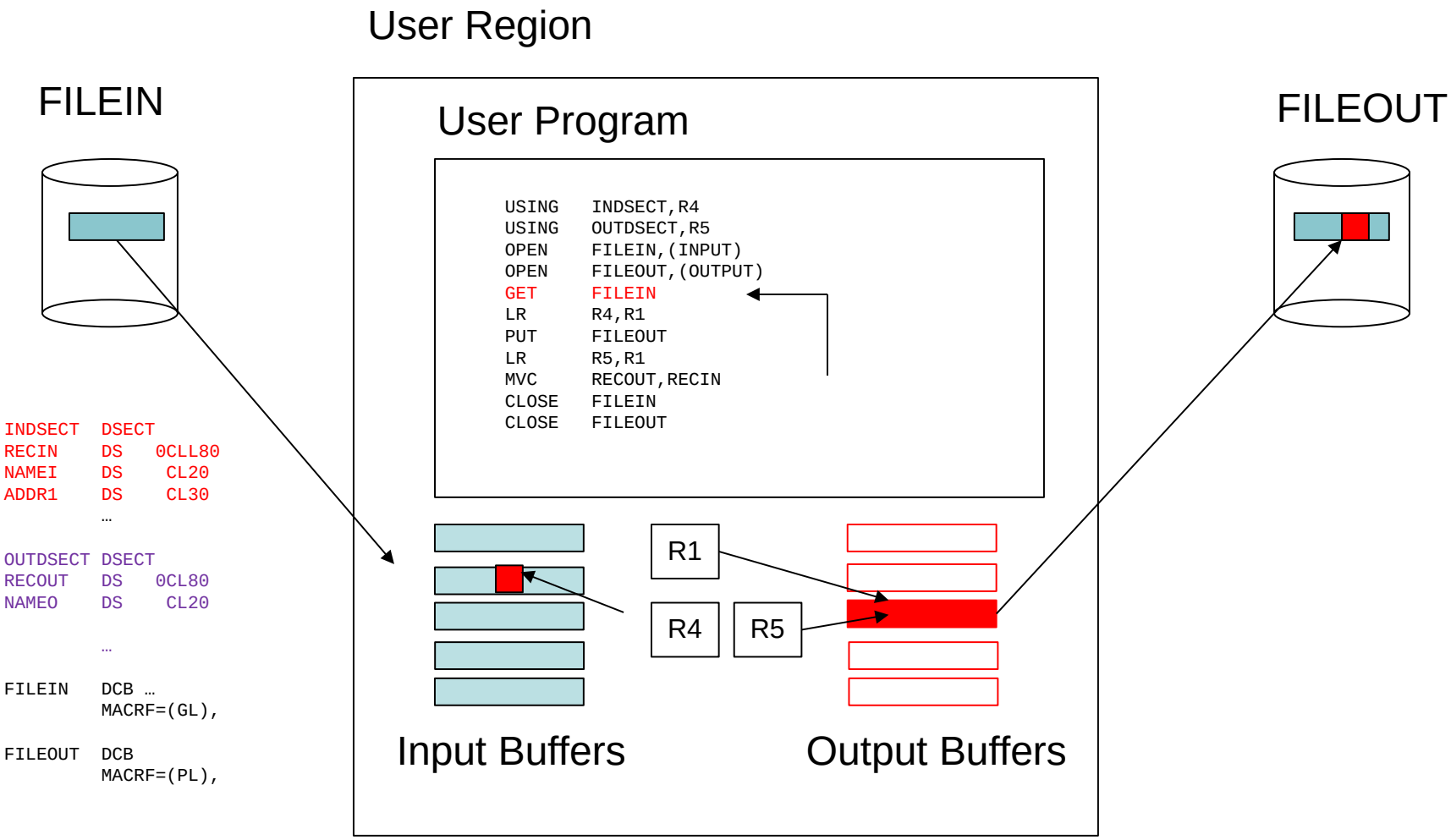
QSAM Locate Mode I/O



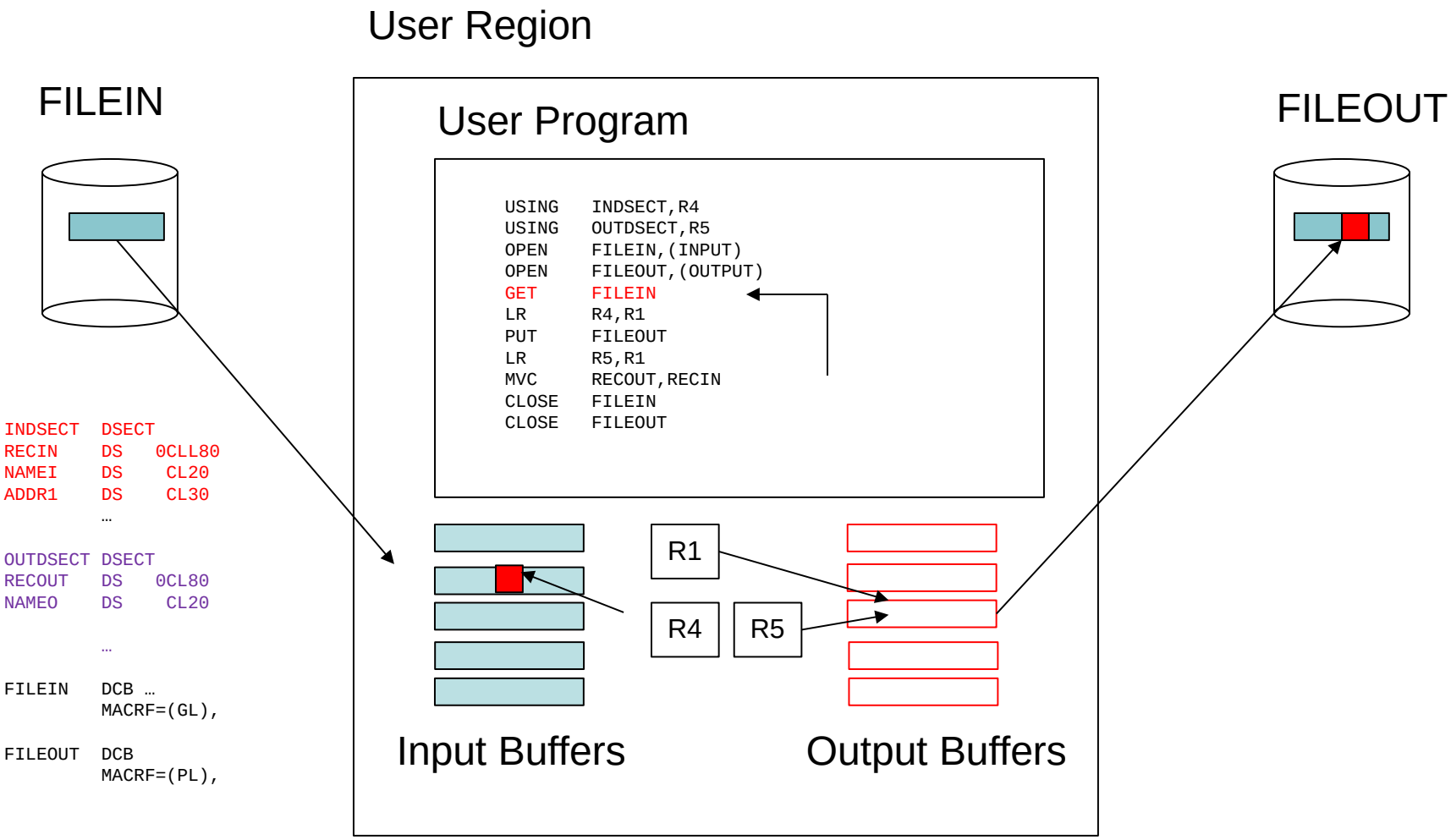
QSAM Locate Mode I/O



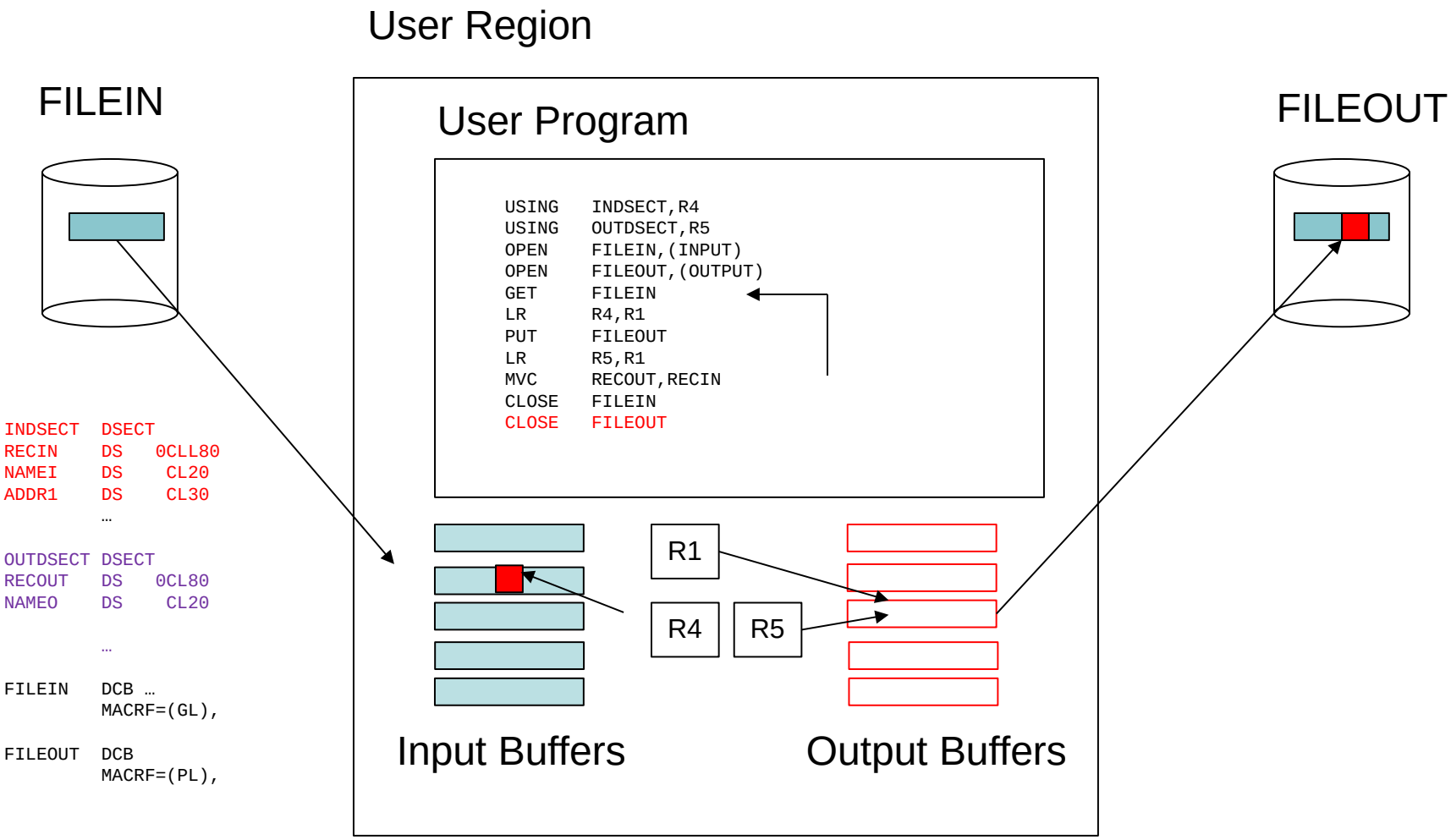
QSAM Locate Mode I/O



QSAM Locate Mode I/O



QSAM Locate Mode I/O



The Using Directive

Example Using

```
USING *,R12
```

Base Address
* = Current Value of Location Counter

Single Base Register

4K range

Domain

Domain starts at Using and Runs to the end of the program

Range starts at the base address and runs for 4k

End of Program

Addressability Rules

- The assembler converts every symbol into a base/displacement address
- Each symbol must be defined in the range of a USING statement (So a displacement can be calculated)
- The use of a symbol must occur in the domain of a USING statement (So a base register can be selected)

Example Using

```
        MVC   X,Y          error
        USING *,R12
        MVC   X,Y
X       DS    CL4
Y       DS    CL4
        …
        MVC   X,Y
```

4K range

Domain

End of Program

Example Using

```
        USING *,R12,R11
        MVC   X,Y
X       DS    CL4
        …
Y       DS    CL4
        …
        MVC   X,Y
        DROP  R12
        MVC   X,Y    ERROR
        MVC   Y,Y    GOOD
```

4K range

4K range

End of Program

USING

- A directive that determines

  1) The base register(s)

  2) The base address from which all displacements are computed
- Op 1 – The base address
- Op 2 – a register (or range) that will be used for base registers

USING Terms

- The domain of a USING directive starts at the USING statement and continues until the end of the program or until all registers mentioned in the USING are dropped
- 

```
USING   *,R12
....
DROP  R12
```

USING Terms

- The range of a USING starts at the base address and continues for 4K
-
```
        USING  *,R12
```
4k

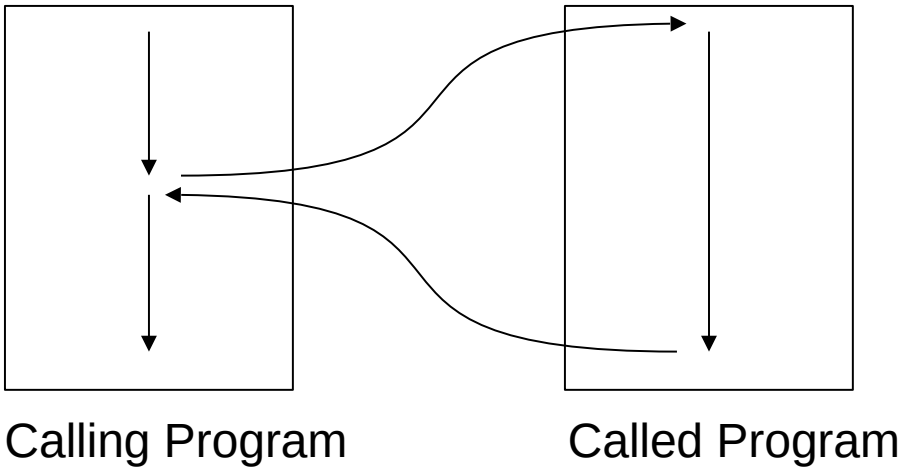
-
```
HERE    EQU   *
          …
          USING HERE,R12
```
4k

Addressability Rules

- Each variable symbol must be defined in the range of a USING statement
- Each use of a variable symbol must occur in the domain of the corresponding USING statement

Linkage

- The process of getting one program to call another



Linkage

- “Linkage conventions” are rules established by IBM for how this happens
- By following linkage conventions, programs written in different languages can cooperate (Cobol calls Assembler)

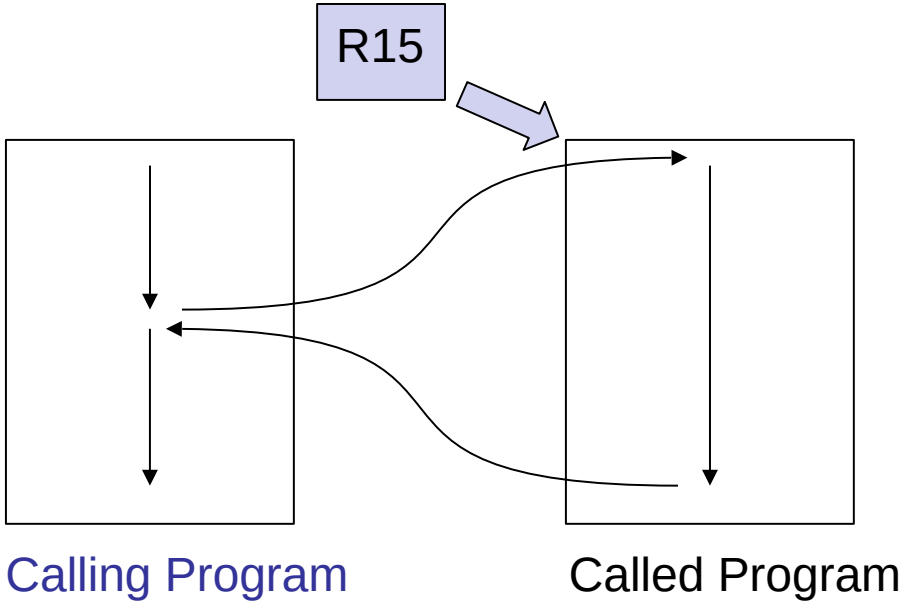
Why Linkage is Important

- Encourages modular design
- When an assembler program is “maxed out” in terms of the number of base registers, it becomes hard to add more code
- One solution is to put new code in a separate module and call it

Linkage Conventions

Calling Program Conventions

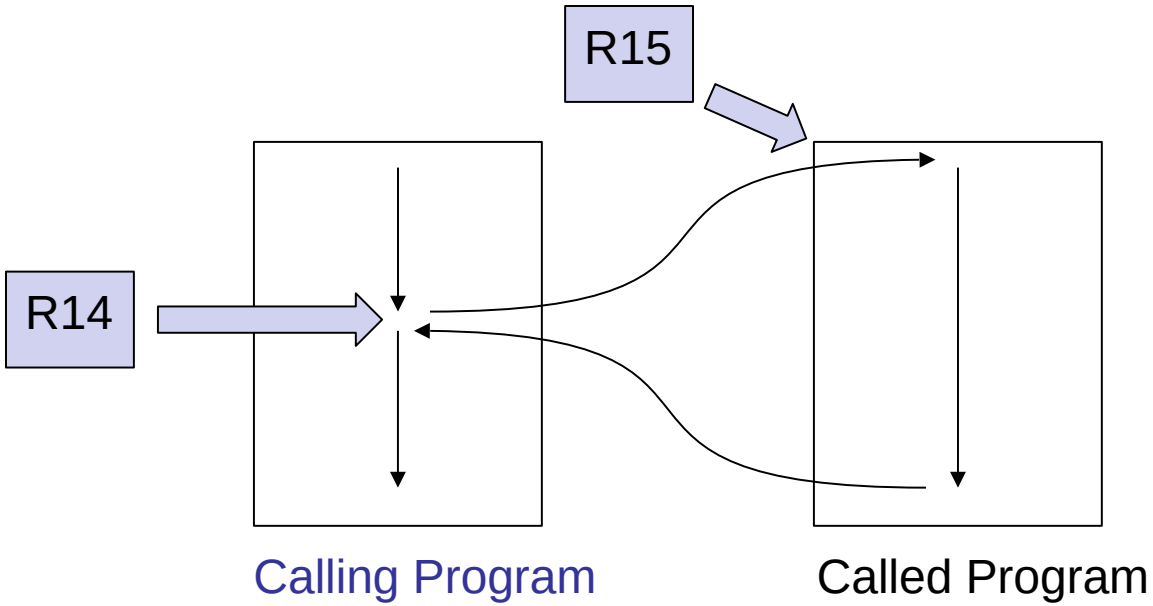
1) Put the address of the target program in R15



Linkage Conventions

Calling Program Conventions

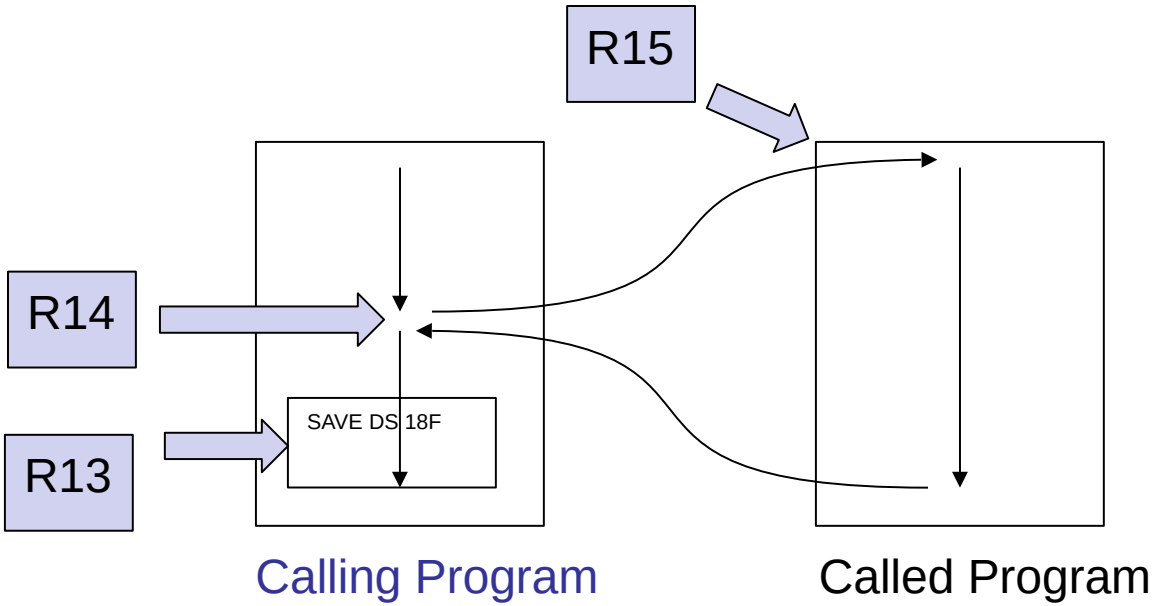
2) Put the return address in R14



Linkage Conventions

Calling Program Conventions

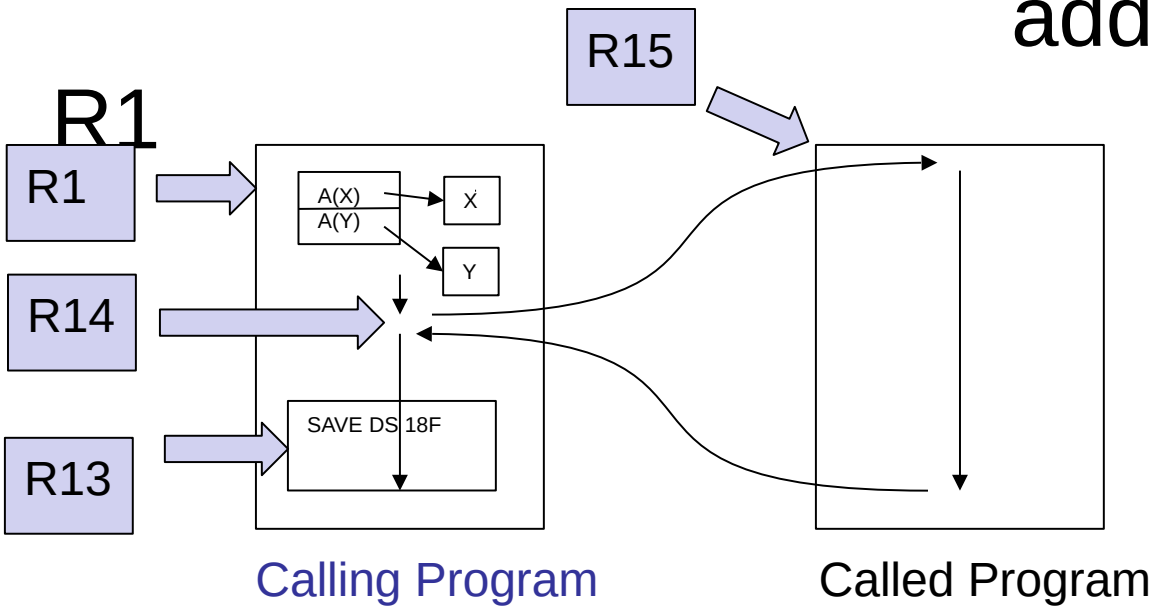
3) Put the address of the save area in R13



Linkage Conventions

Calling Program Conventions

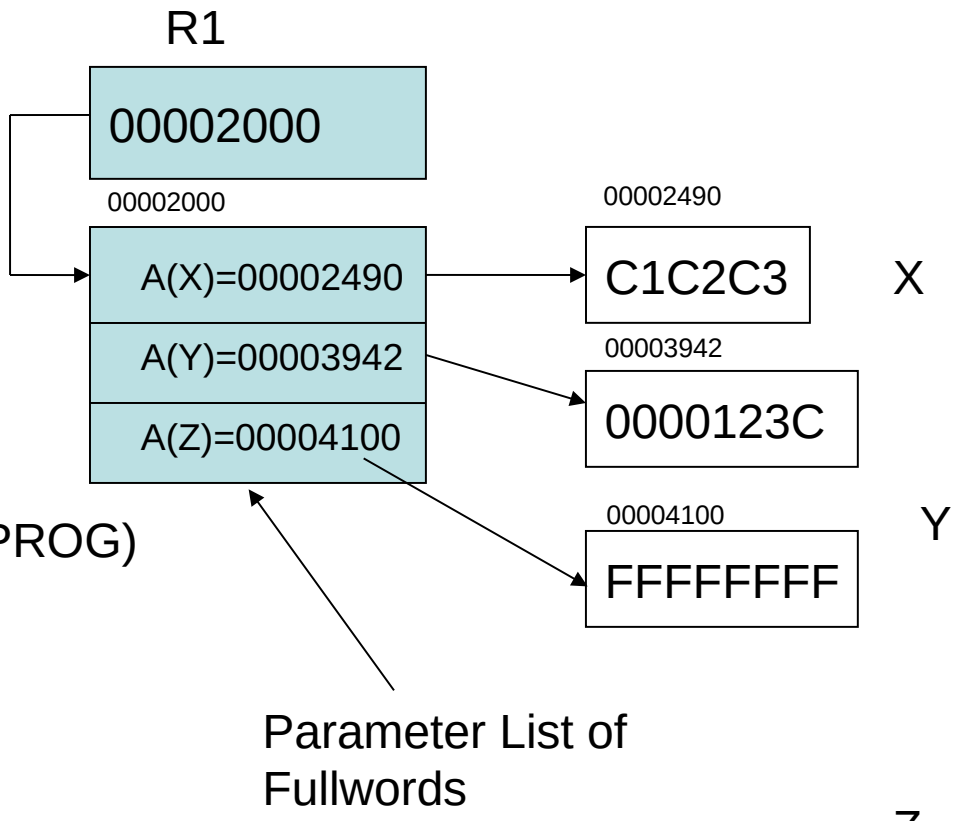
4) Put the address of the list of parameter addresses in R1



Passing Parms

The calling program wants to pass X, Y, and Z

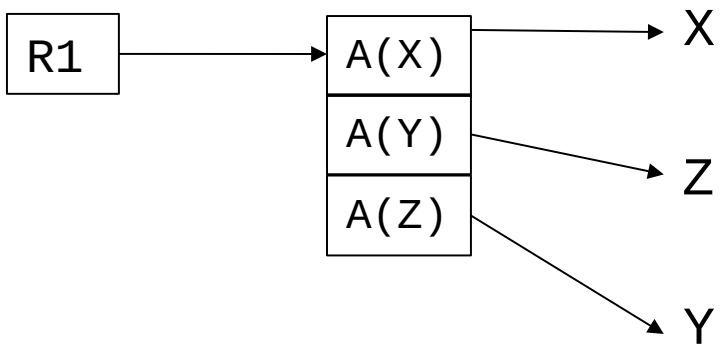
```
X     DS   CL3
      ...
Y     DS   PL4
      ...
Z     DS   F

LA   R1,=A(X,Y,Z)
L    R15,=V(SUBPROG)
BASR  R14,R15
```



Calling Program Code

```
        LA    R13,SAVE
        …
        LA    R1,=A(X,Y,Z)
        L     R15,=V(TARGET)
        BASR  R14,R15
```

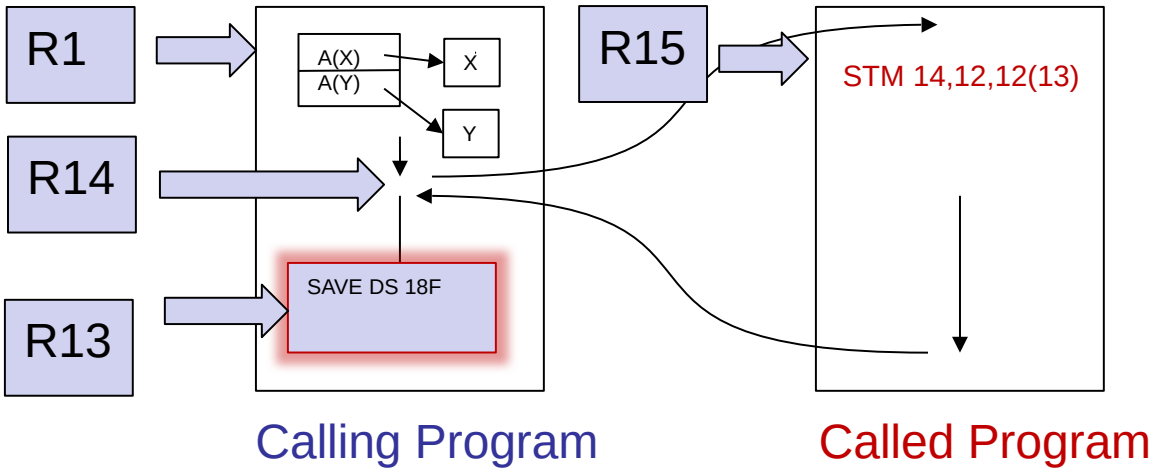
```
SAVE  DS  18F
X     DS  CL3
…
Z     DS  F
Y     DS  PL4
```



Linkage Conventions

Called Program Conventions

1) Store the calling programs registers in the calling program's save area

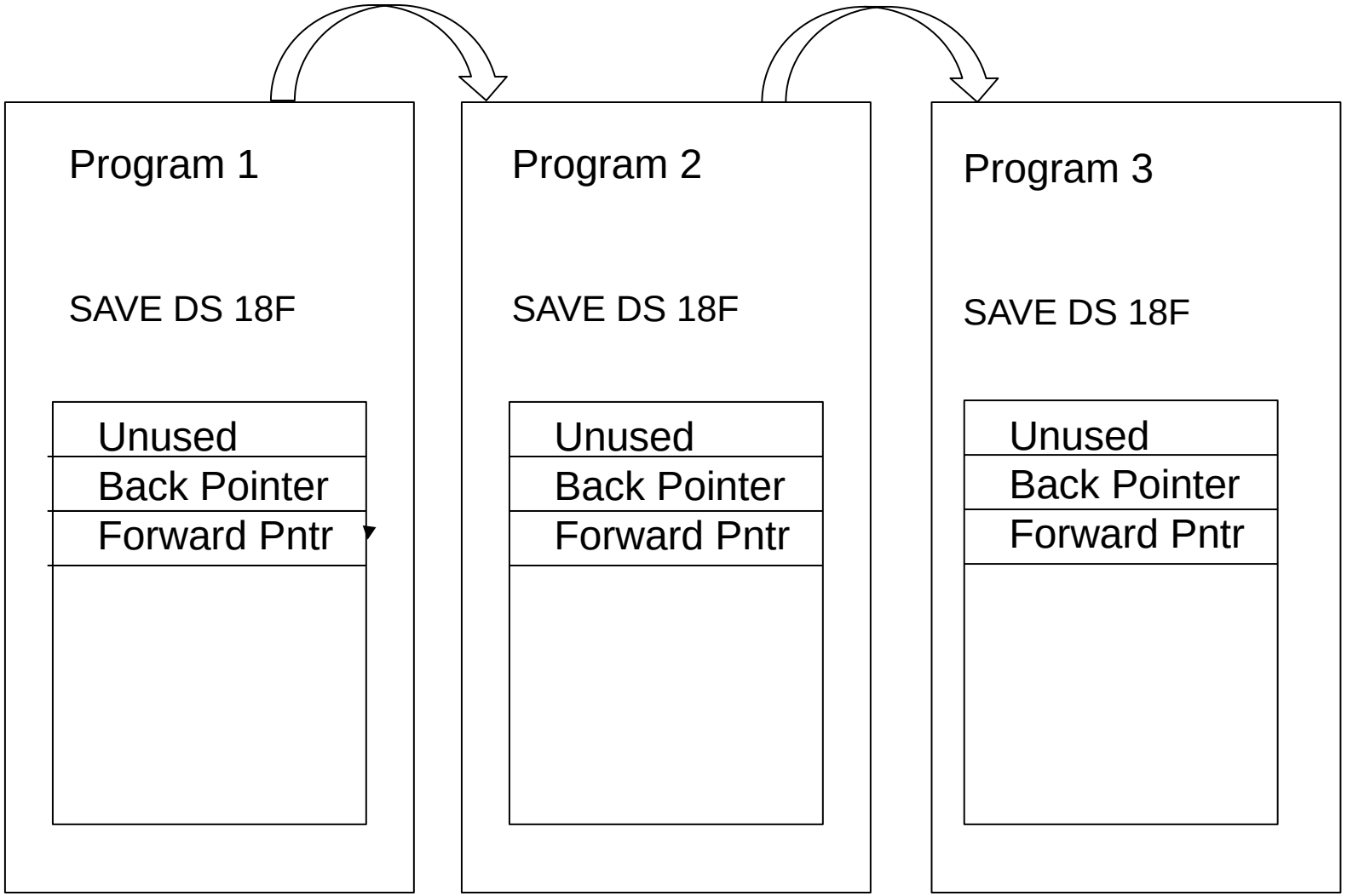


Save Area Format

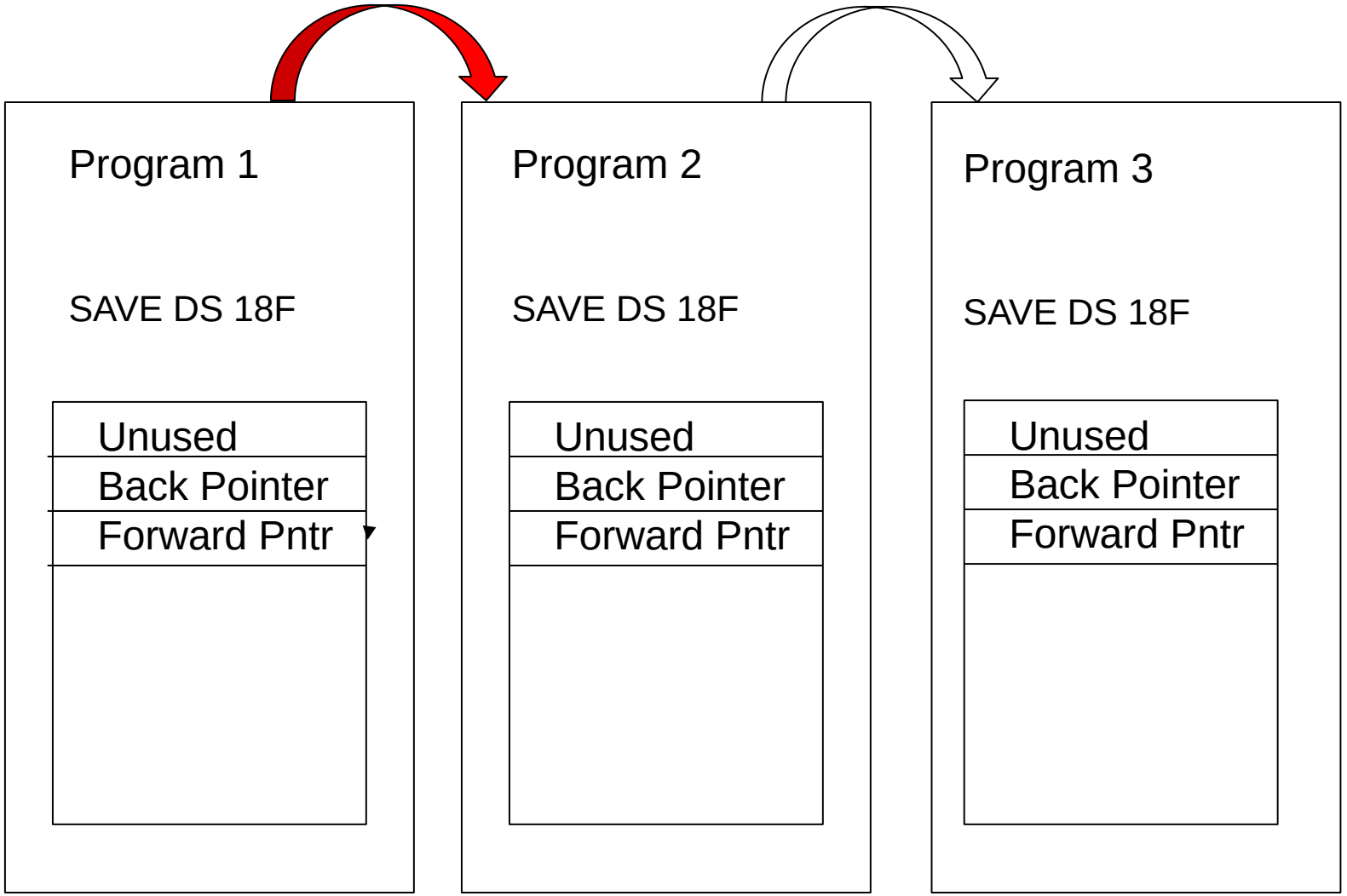
Word	Contents
0	Used by PL/I, if applicable. Otherwise, unused.
1	If applicable, the address of the calling program's register save area.
2	The address of the current program's next register save area.
3	The contents of register 14 (the return address within the calling program).
4	The contents of register 15 (the address of the called program).
5	The contents of register 0.
6	The contents of register 1.
7	The contents of register 2.
8	The contents of register 3.
9	The contents of register 4.
10	The contents of register 5.
11	The contents of register 6.
12	The contents of register 7.
13	The contents of register 8.
14	The contents of register 9.
15	The contents of register 10.
16	The contents of register 11.

```
SAVE     DS    18F
```

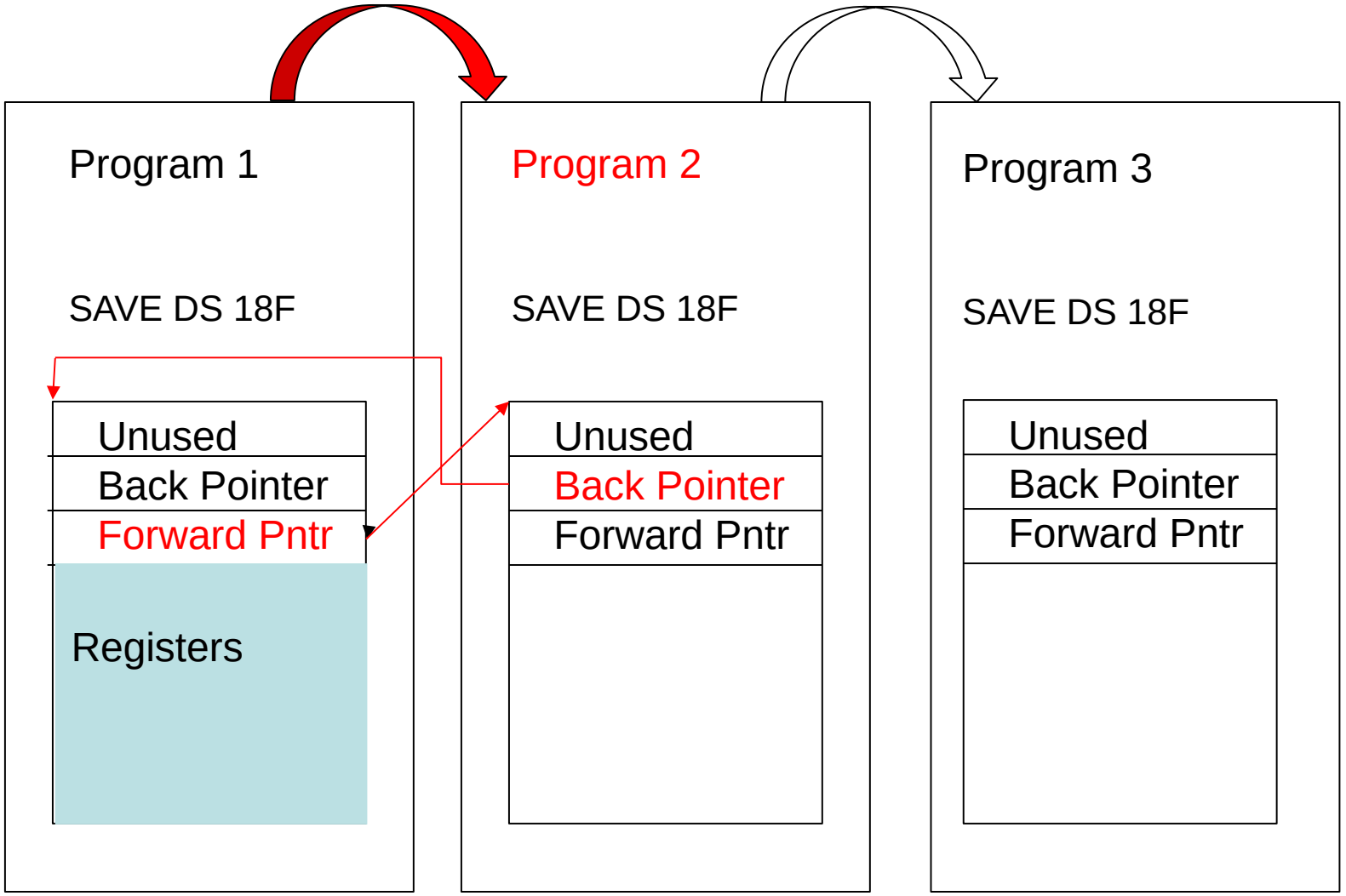
Save Area Chain



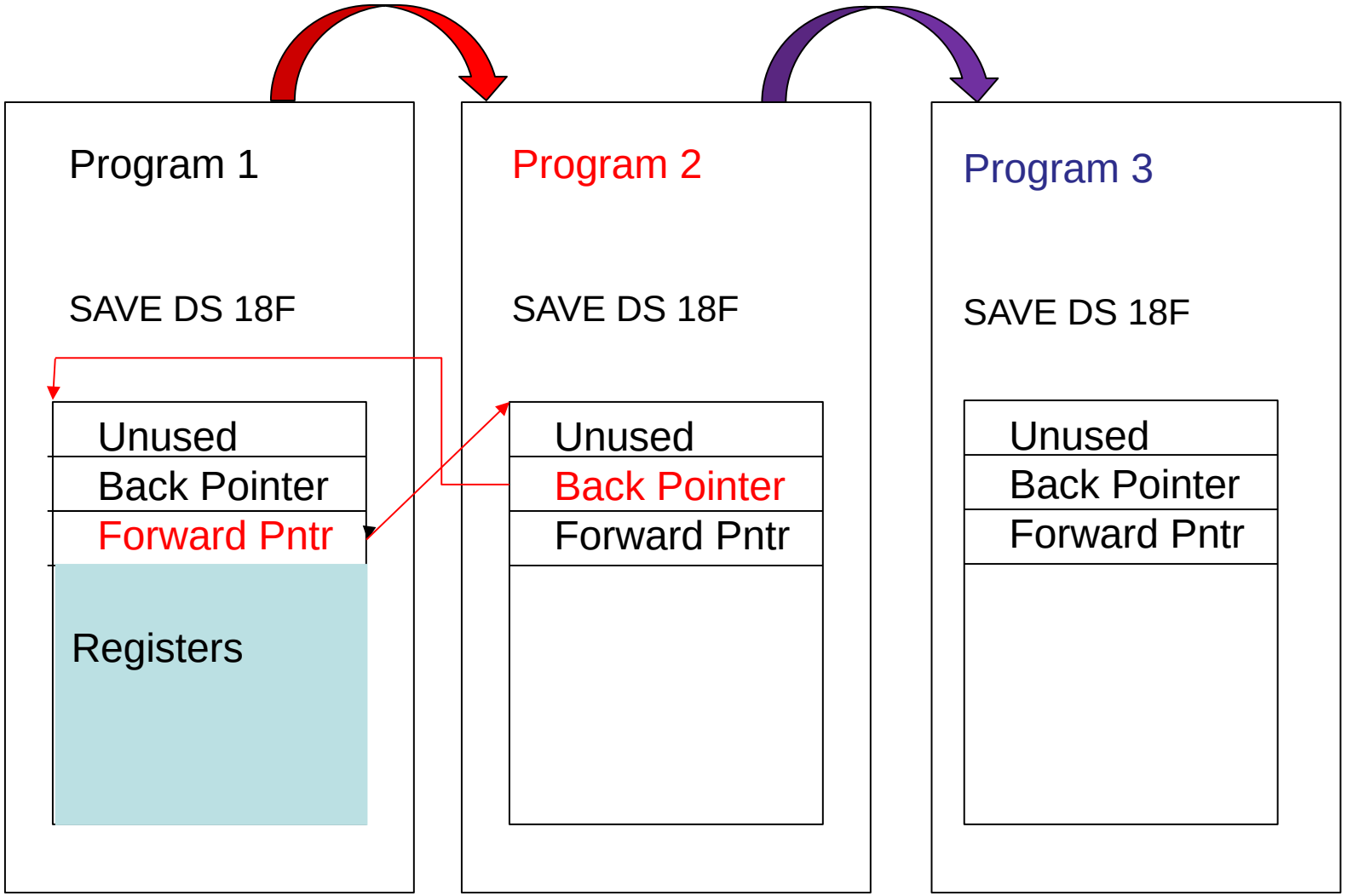
Save Area Chain



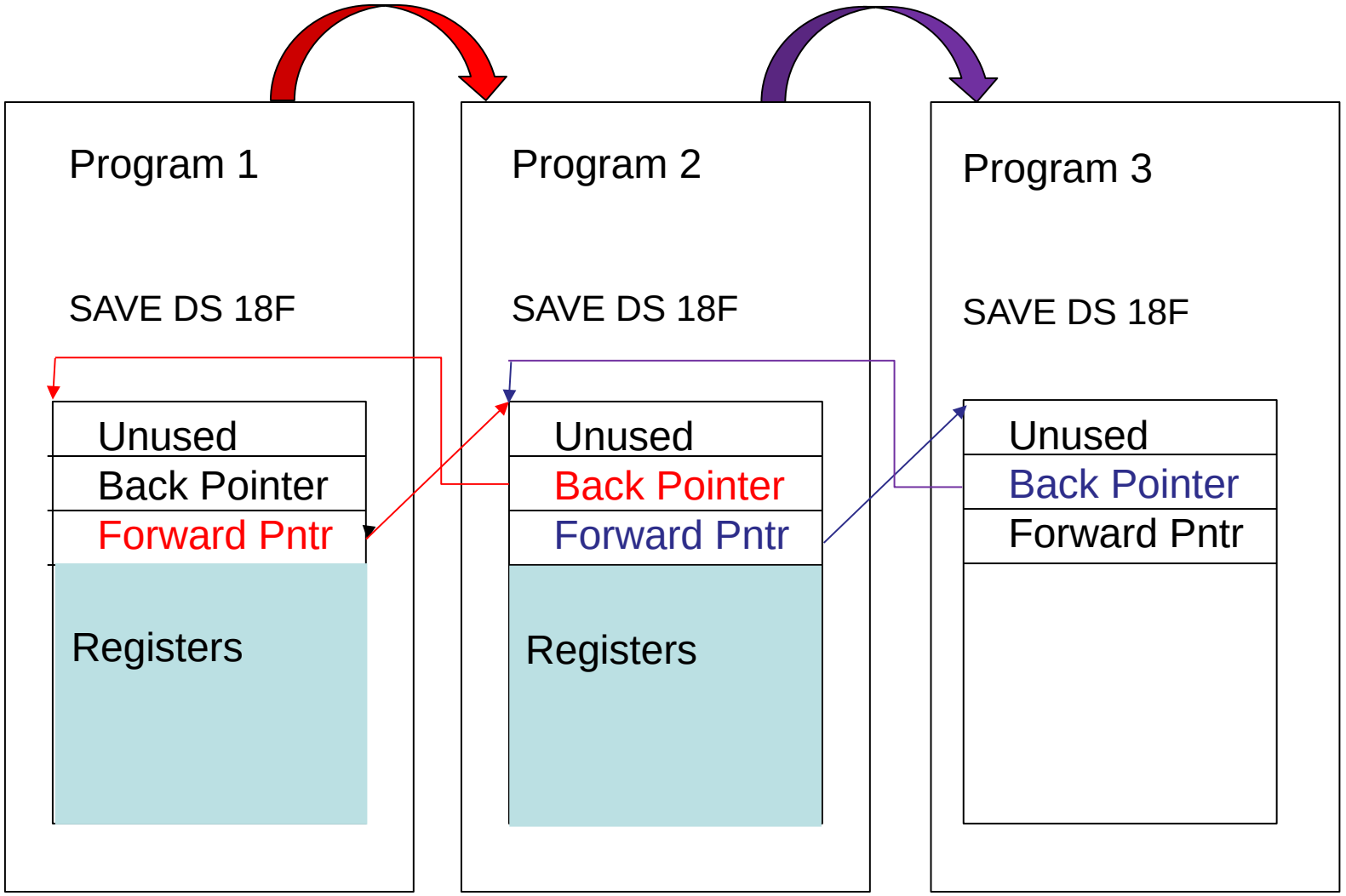
Save Area Chain



Save Area Chain



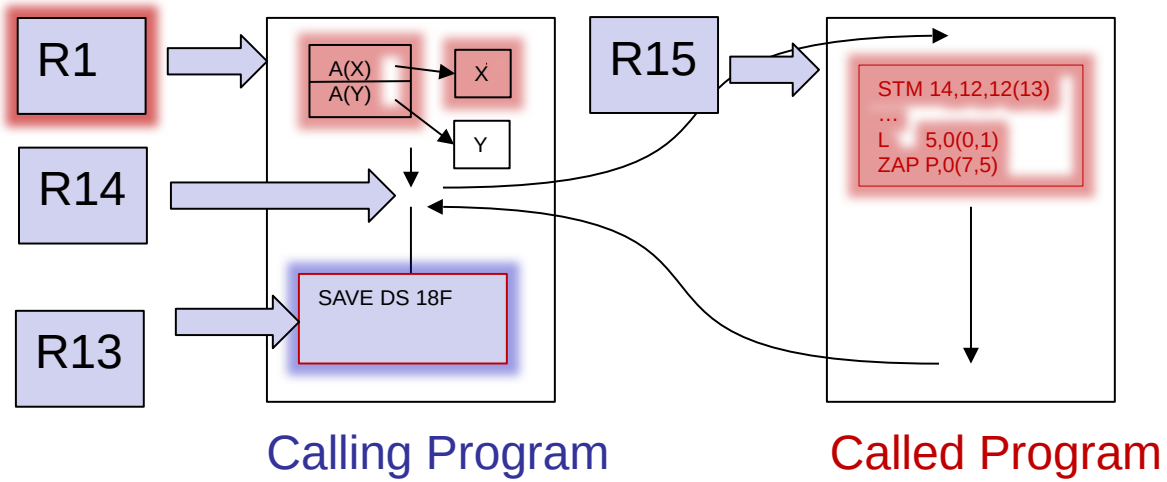
Save Area Chain



Linkage Conventions

Called Program Conventions

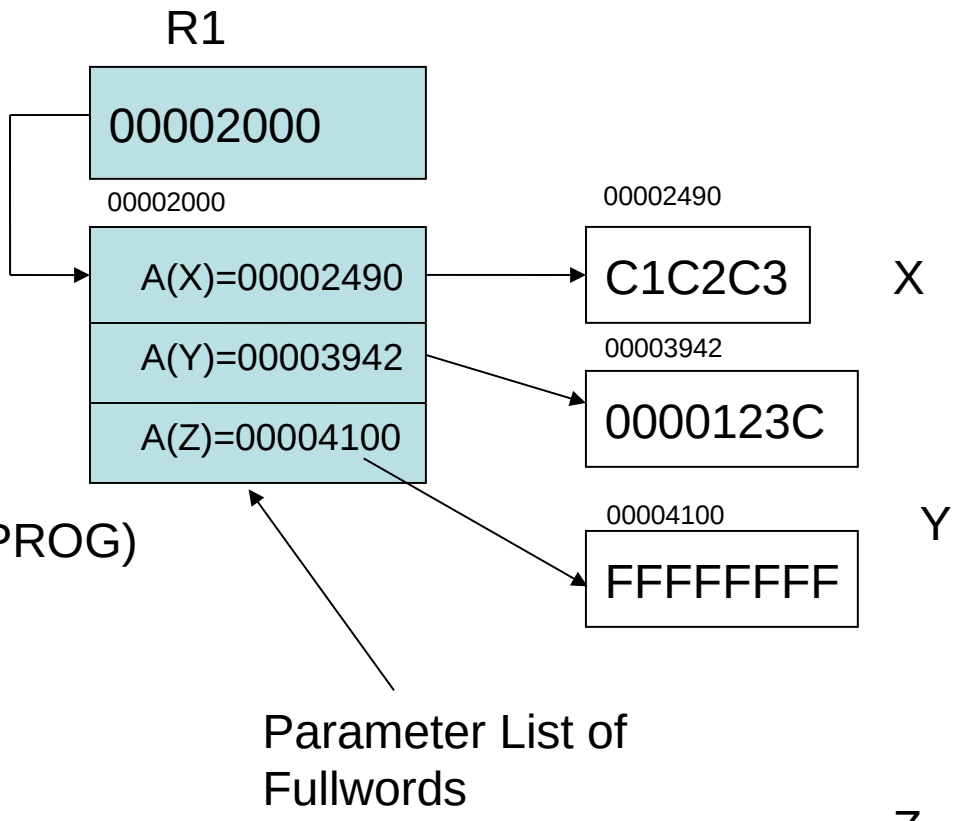
2) Use R1 to access the parameters that were passed



Passing Parms

The calling program wants to pass X, Y, and Z

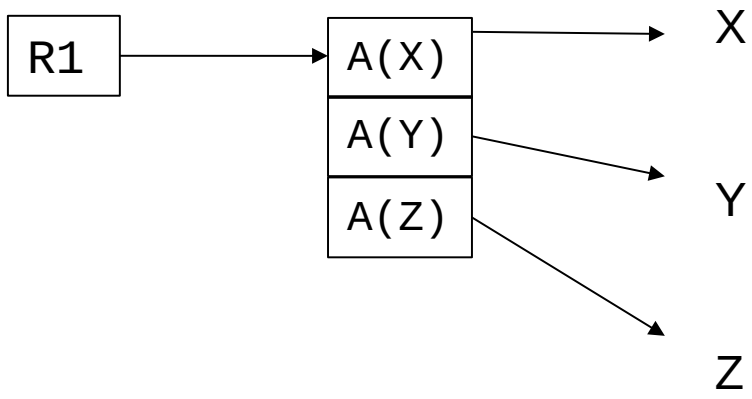
```
X     DS   CL3
      ...
Y     DS   PL4
      ...
Z     DS   F

LA   R1,=A(X,Y,Z)
L    R15,=V(SUBPROG)
BASR  R14,R15
```



Called Program Code

```
STM    R14,R12,12(R13)
…
ST     R13,SAVE+4
LA     R13,SAVE
…
L      R5,0(R0,R1)
ZAP    XSUB,0(L'XSUB,R5)
L      R5,4(R0,R1)
MVC    YSUB,0(R5)
…
L      R5,8(R0,R1)
MVC    0(L'ANSWER,R5),ANSWER
…
L      R0,ANSWER
LA     R15,0
BR     R14
```

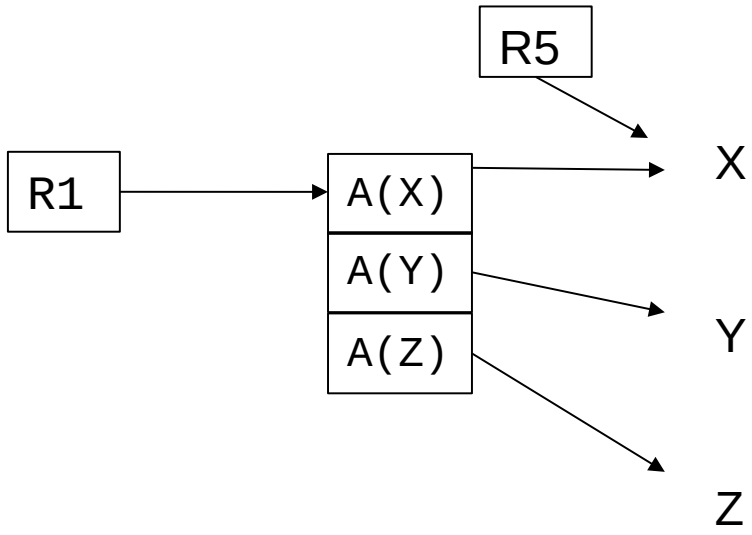
```
SAVE     DS 18F
XSUB     DS PL3
YSUB     DS CL4

ANSWER   DS F
```



Called Program Code

```
STM     R14,R12,12(R13)
…
ST      R13,SAVE+4
LA      R13,SAVE
…
L       R5,0(R0,R1)
ZAP     XSUB,0(L'XSUB,R5)
L       R5,4(R0,R1)
MVC     YSUB,0(R5)
…
L       R5,8(R0,R1)
MVC     0(L'ANSWER,R5),ANSWER
…
L       R0,ANSWER
LA      R15,0
BR      R14
```

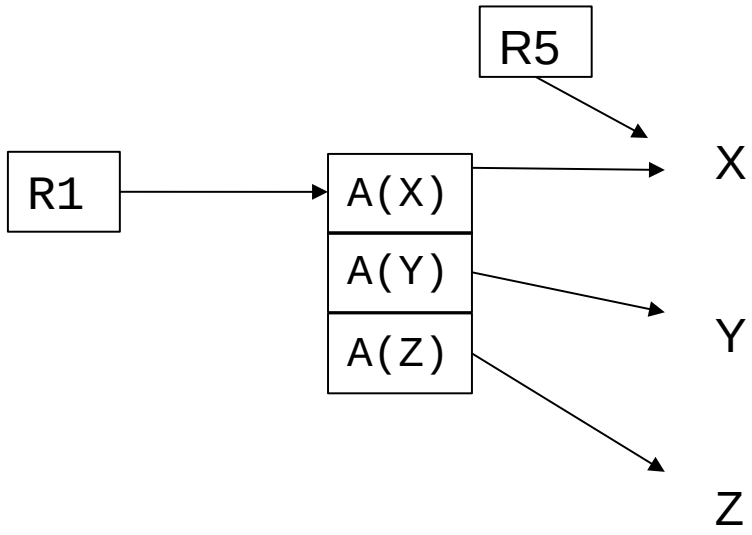
```
SAVE    DS  18F
XSUB    DS  PL3
YSUB    DS  CL4

ANSWER  DS  F
```



Called Program Code

```
STM     R14,R12,12(R13)
…
ST      R13,SAVE+4
LA      R13,SAVE
…
L       R5,0(R0,R1)
ZAP     XSUB,0(L'XSUB,R5)
L       R5,4(R0,R1)
MVC     YSUB,0(R5)
…
L       R5,8(R0,R1)
MVC     0(L'ANSWER,R5),ANSWER
…
L       R0,ANSWER
LA      R15,0
BR      R14
```

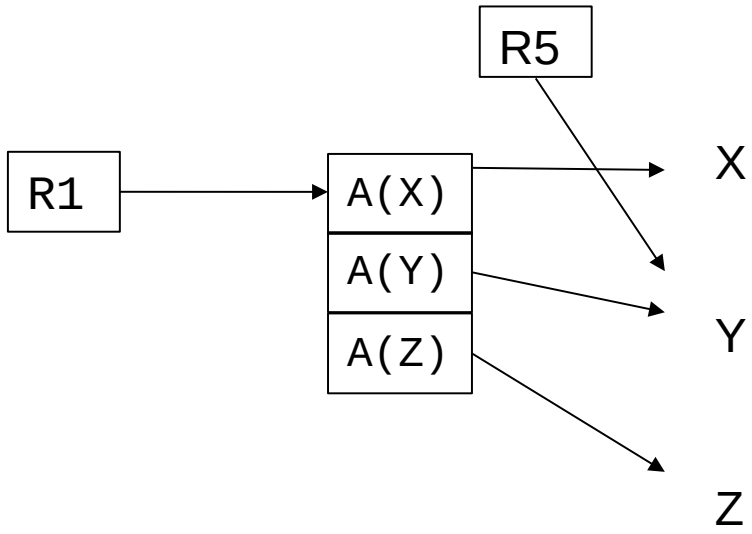
```
SAVE    DS  18F
XSUB    DS  PL3
YSUB    DS  CL4

ANSWER  DS  F
```



Called Program Code

```
STM     R14,R12,12(R13)
…
ST      R13,SAVE+4
LA      R13,SAVE
…
L       R5,0(R0,R1)
ZAP     XSUB,0(L'XSUB,R5)
L       R5,4(R0,R1)
MVC     YSUB,0(R5)
…
L       R5,8(R0,R1)
MVC     0(L'ANSWER,R5),ANSWER
…
L       R0,ANSWER
LA      R15,0
BR      R14
```

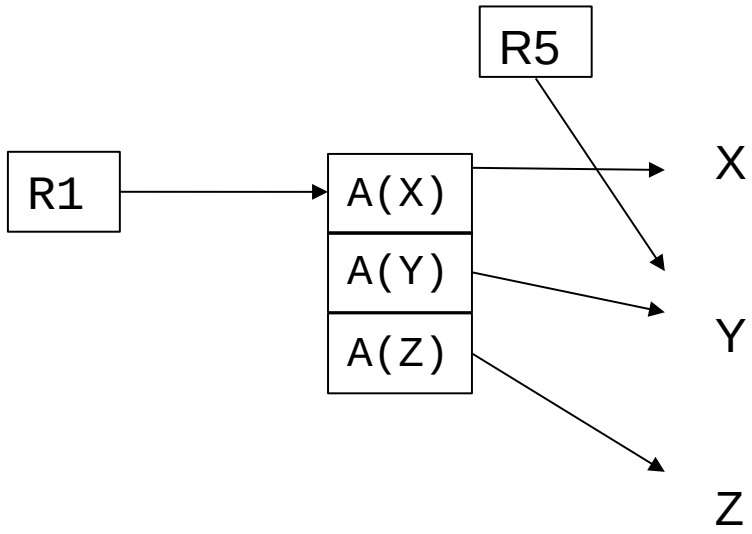
```
SAVE    DS  18F
XSUB    DS  PL3
YSUB    DS  CL4

ANSWER  DS  F
```



Called Program Code

```
STM     R14,R12,12(R13)
…
ST      R13,SAVE+4
LA      R13,SAVE
…
L       R5,0(R0,R1)
ZAP     XSUB,0(L'XSUB,R5)
L       R5,4(R0,R1)
MVC     YSUB,0(R5)
…
L       R5,8(R0,R1)
MVC     0(L'ANSWER,R5),ANSWER
…
L       R0,ANSWER
LA      R15,0
BR      R14
```

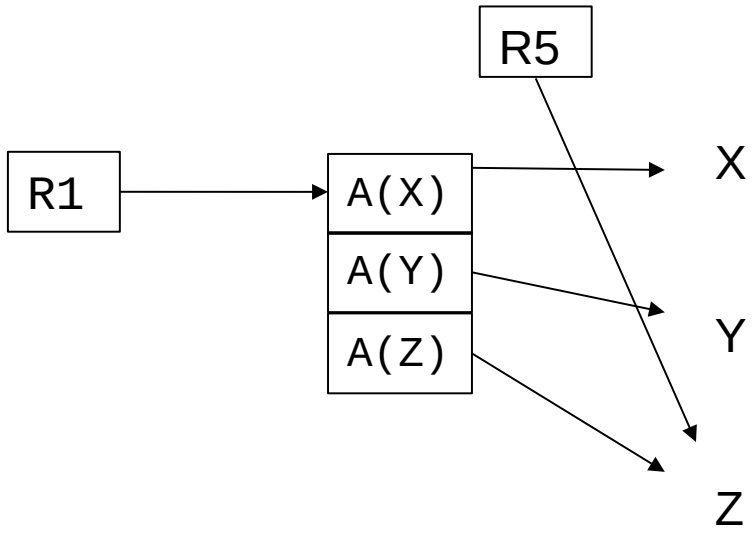
```
SAVE    DS  18F
XSUB    DS  PL3
YSUB    DS  CL4

ANSWER  DS  F
```



Called Program Code

```
STM     R14,R12,12(R13)
…
ST      R13,SAVE+4
LA      R13,SAVE
…
L       R5,0(R0,R1)
ZAP     XSUB,0(L'XSUB,R5)
L       R5,4(R0,R1)
MVC     YSUB,0(R5)
…
L       R5,8(R0,R1)
MVC     0(L'ANSWER,R5),ANSWER
…
L       R0,ANSWER
LA      R15,0
BR      R14
```

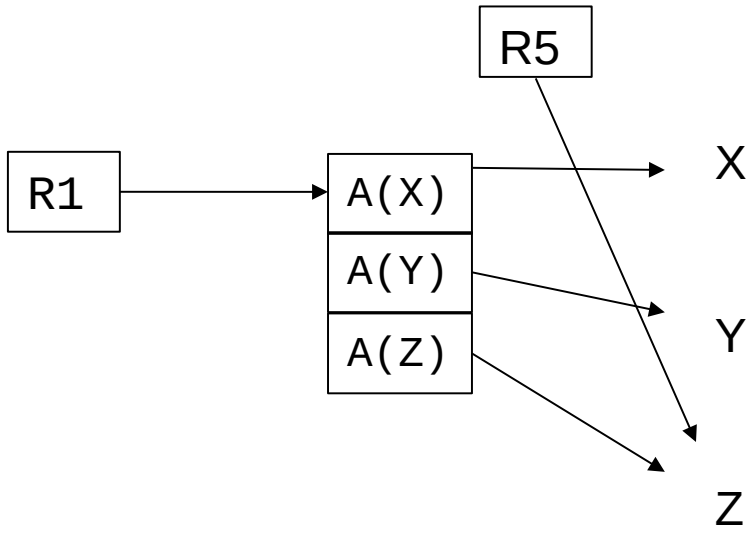
```
SAVE    DS  18F
XSUB    DS  PL3
YSUB    DS  CL4

ANSWER  DS  F
```



Called Program Code

```
STM     R14,R12,12(R13)
…
ST      R13,SAVE+4
LA      R13,SAVE
…
L       R5,0(R0,R1)
ZAP     XSUB,0(L'XSUB,R5)
L       R5,4(R0,R1)
MVC     YSUB,0(R5)
…
L       R5,8(R0,R1)
MVC     0(L'ANSWER,R5),ANSWER
…
L       R0,ANSWER
LA      R15,0
BR      R14
```

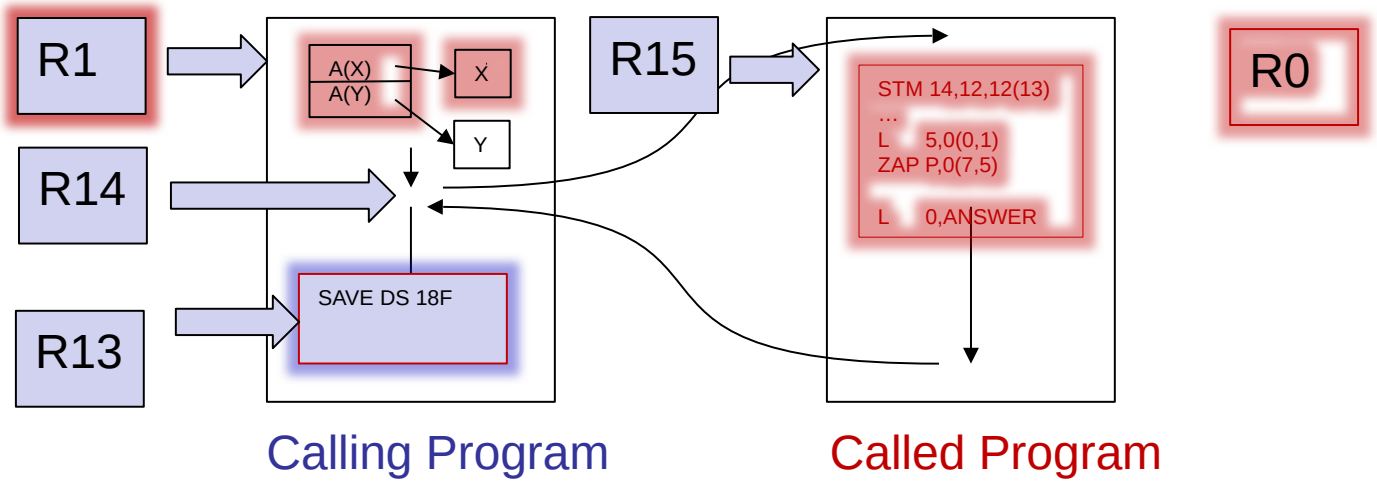
```
SAVE    DS  18F
XSUB    DS  PL3
YSUB    DS  CL4

ANSWER  DS  F
```



Linkage Conventions

Called Program Conventions

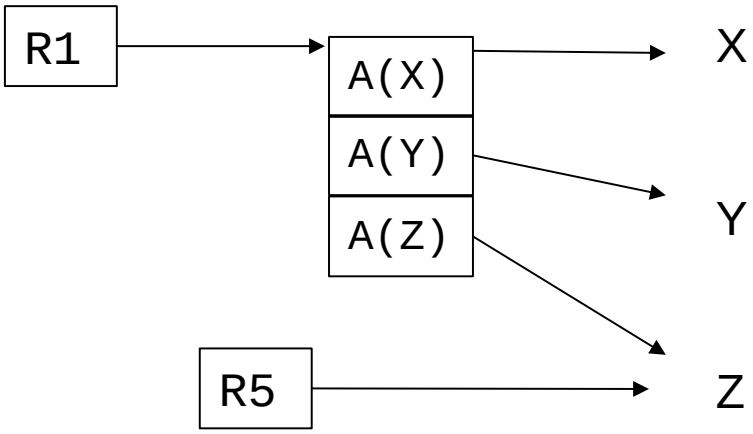
3) For a function subprogram, return the answer in R0



Called Program Code

```
STM     R14,R12,12(R13)
…
ST      R13,SAVE+4
LA      R13,SAVE
…
L       R5,0(R0,R1)
ZAP     XSUB,0(L'XSUB,R5)
L       R5,4(R0,R1)
MVC     YSUB,0(R5)
…
L       R5,8(R0,R1)
MVC     0(L'ANSWER,R5),ANSWER
…
L       R0,ANSWER
LA      R15,0
BR      R14
```

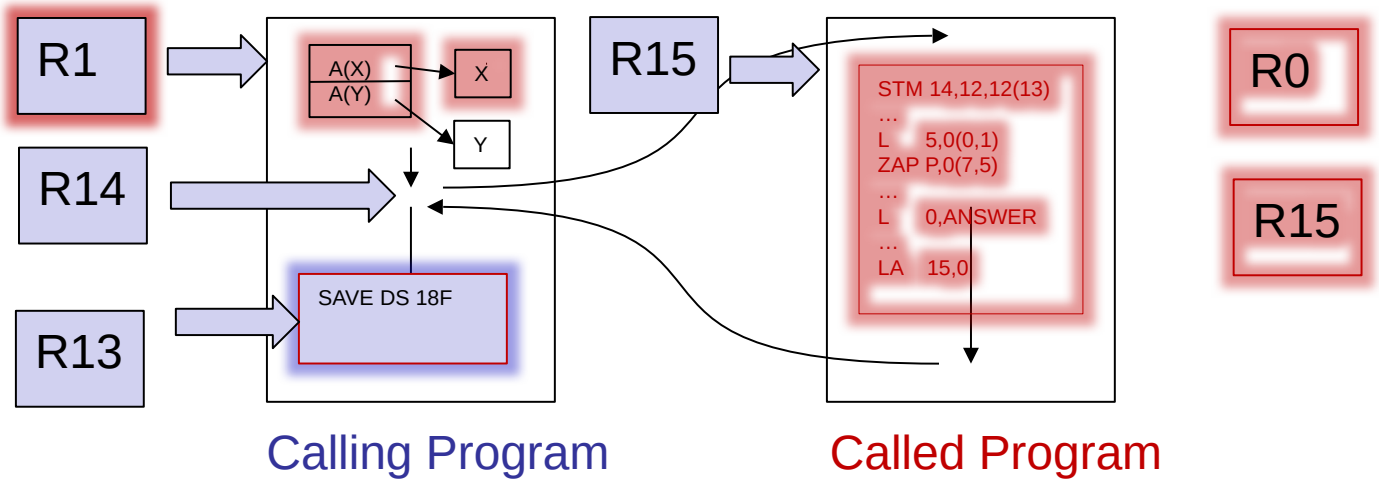
```
SAVE    DS  18F
XSUB    DS  PL3
YSUB    DS  CL4

ANSWER  DS  F
```



Linkage Conventions

Called Program Conventions

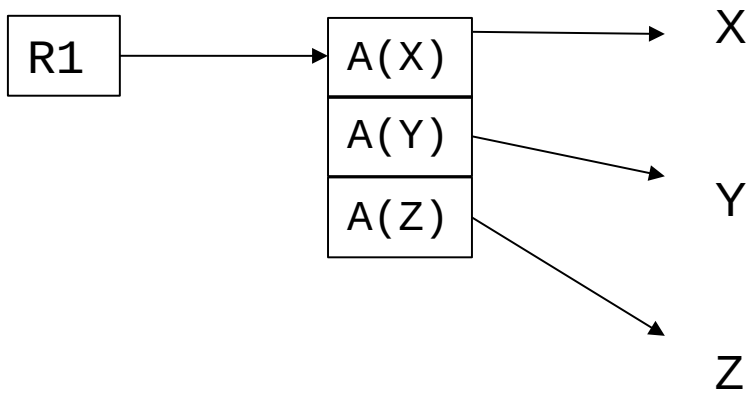
4) Put a return code in R15



Called Program Code

```
STM    R14,R12,12(R13)
…
ST     R13,SAVE+4
LA     R13,SAVE
…
L      R5,0(R0,R1)
ZAP    XSUB,0(L'XSUB,R5)
L      R5,4(R0,R1)
MVC    YSUB,0(R5)
…
L      R5,8(R0,R1)
MVC    0(L'ANSWER,R5),ANSWER
…
L      R0,ANSWER
LA     R15,0
BR     R14
```

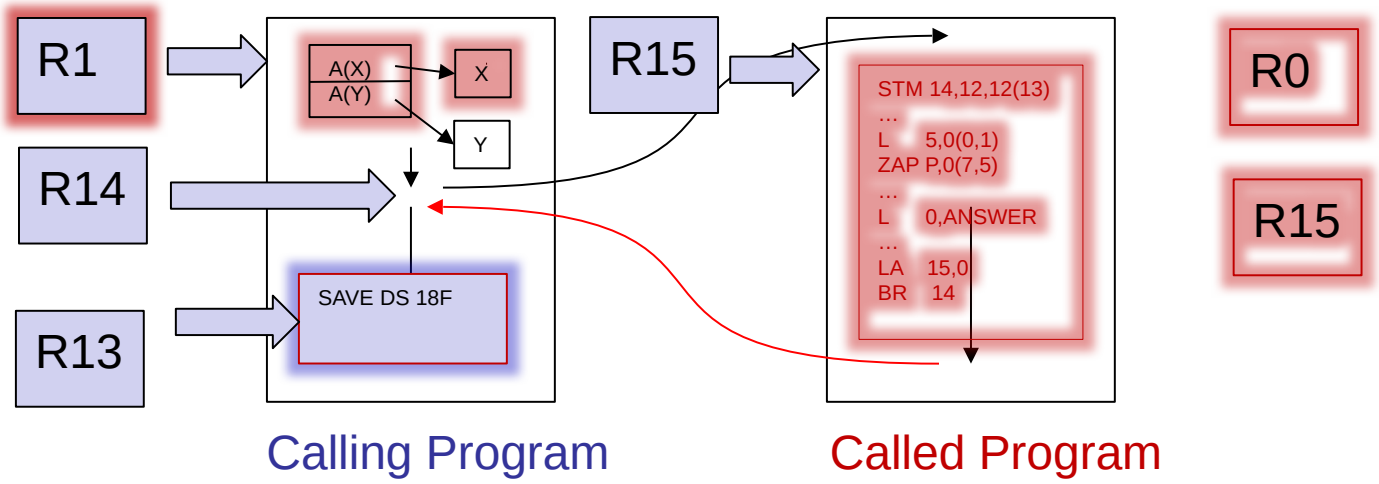
```
SAVE     DS 18F
XSUB     DS PL3
YSUB     DS CL4

ANSWER   DS F
```



Linkage Conventions

Called Program Conventions

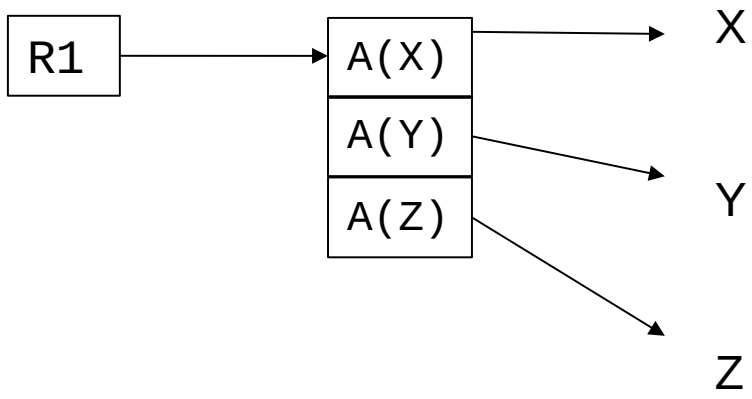
5) Return to the caller on R14



Called Program Code

```
STM    R14,R12,12(R13)
…
ST     R13,SAVE+4
LA     R13,SAVE
…
L      R5,0(R0,R1)
ZAP    XSUB,0(L'XSUB,R5)
L      R5,4(R0,R1)
MVC    YSUB,0(R5)
…
L      R5,8(R0,R1)
MVC    0(L'ANSWER,R5),ANSWER
…
L      R0,ANSWER
LA     R15,0
BR     R14
```

```
SAVE     DS 18F
XSUB     DS PL3
YSUB     DS CL4

ANSWER   DS F
```



Summary: Linkage Conventions

Calling Program Conventions

1) Put the address of the target program in R15
2) Put the return address in R14
3) Put the address of the save area in R13
4) Put the address of the list of parameter addresses in R1

Summary: Linkage Conventions

Called Program Conventions

1) Store the calling programs registers in the calling program's save area
2) Use R1 to access the parameters that were passed
3) For a function subprogram, return the answer in R0
4) Put a return code in R15
5) Return to the caller on R14

The Calling Chain

- The Operating System is the “ultimate” caller that treats your program like a subprogram

The Calling Chain

- Your program can act as a caller for another

The Calling Chain

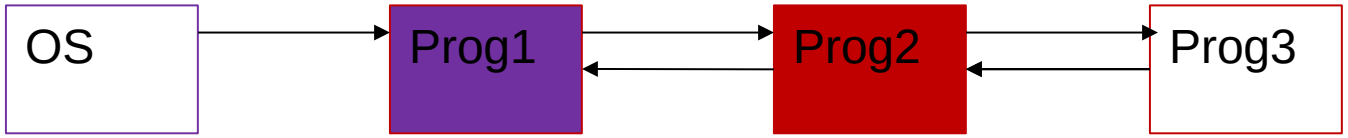
- Which can call another program …

The Calling Chain

- Eventually a program completes and returns to its caller …

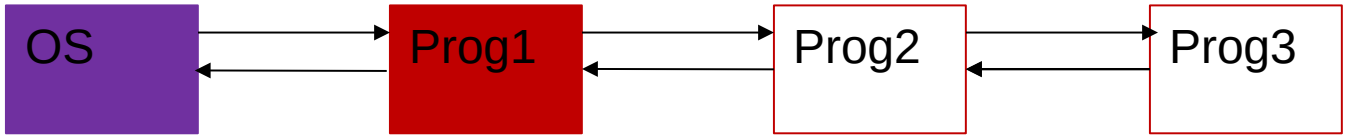
The Calling Chain

- That caller acts as called program and it, too, returns …



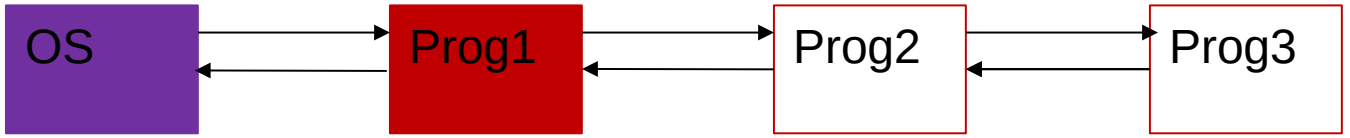
The Calling Chain

- All the way up the chain …



The Calling Chain

- Each program in the chain may have two faces: caller and called



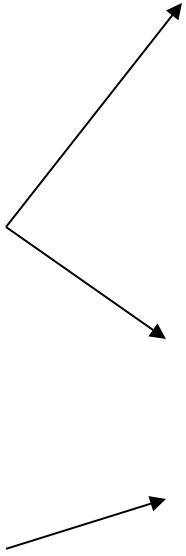
- Lets examine the standard entry and exit code to see how these duties are completed

Standard Entry Code (Partial)

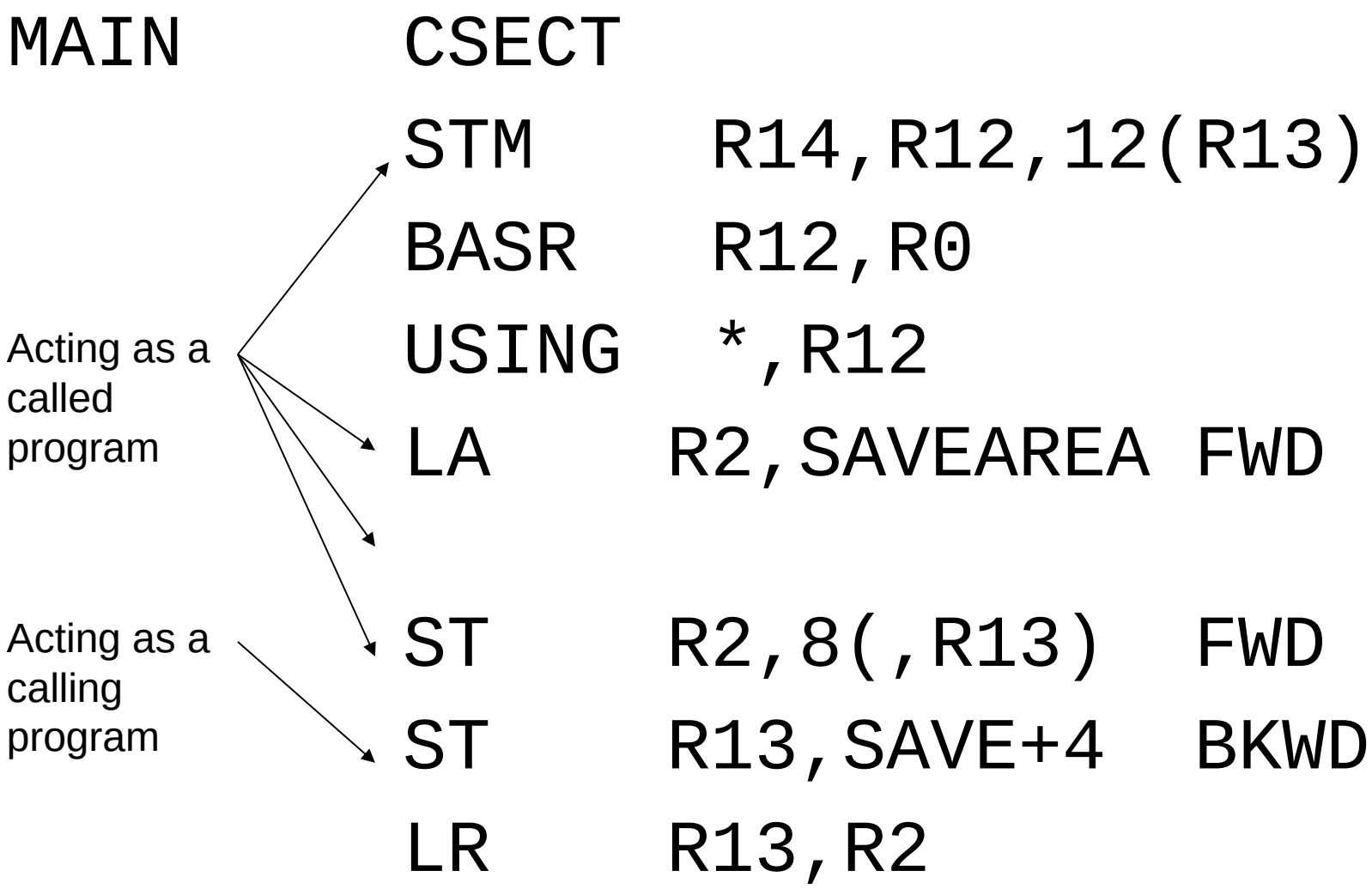
```
MAIN     CSECT
         STM    R14,R12,12(R13)
         BASR   R12,R0
         USING  *,R12
         ST     R13,SAVE+4
         LA     R13,SAVE
```

Acting as a called program

Acting as a calling program



Standard Entry Code (Full)

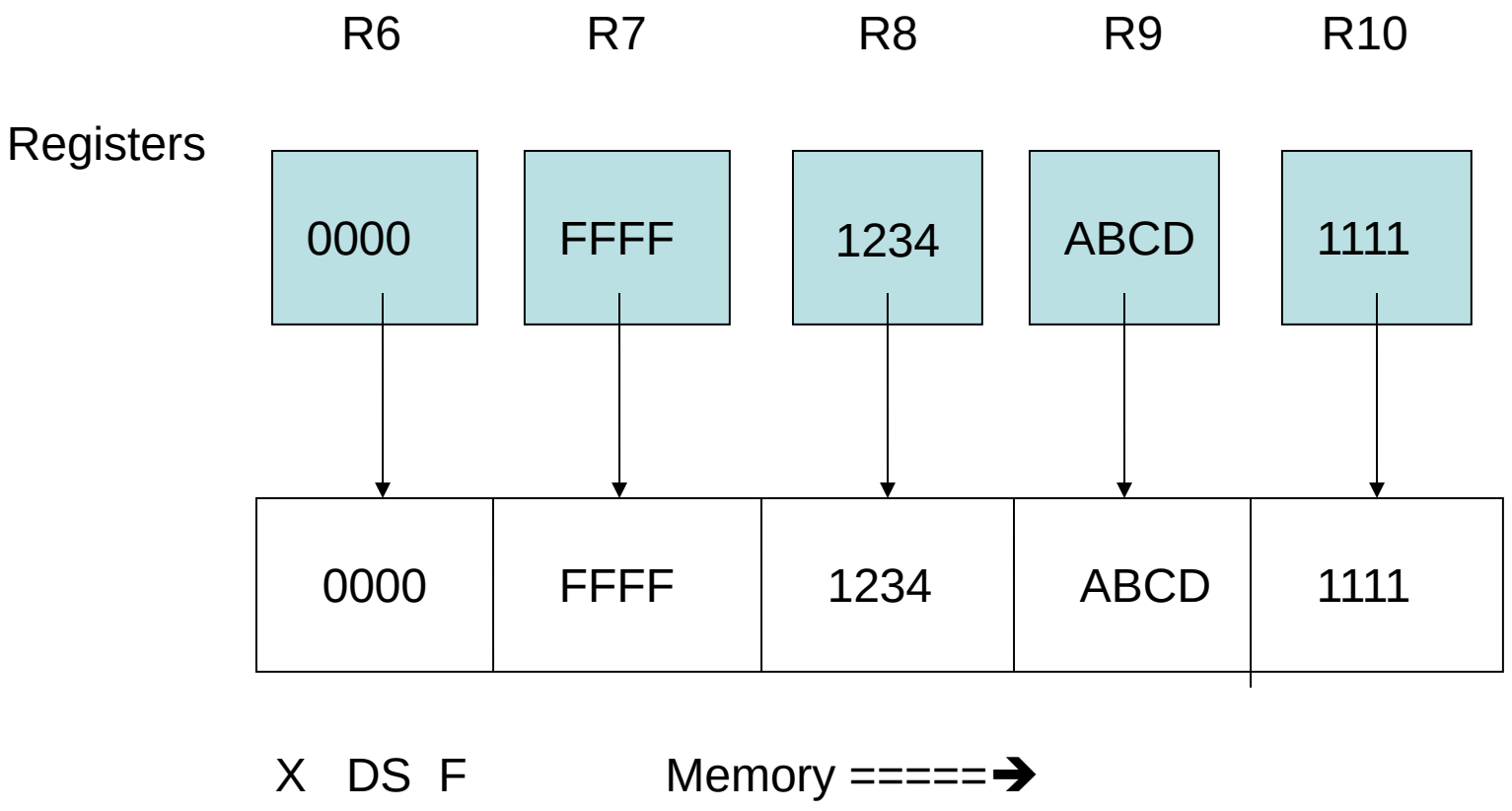


Store Multiple

- STM
- RS
- Op 1 – Beginning register of a range
- Op 2 – Ending register of a range
- Op 3 – a Fullword in memory
- Consecutive fullwords starting at Op-3 are loaded into consecutive registers beginning with Op 1.

STM

STM R6,R10,X

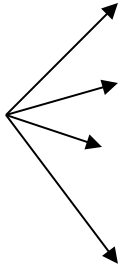


BASR

- BASR – Branch and Save Register
- Op 1 – A register
- Op 2 – A register
- The address of the next instruction is stored in Op 1, and a branch occurs to the address in Op 2.
- If Op 2 is Register 0, no branch occurs

Standard Exit Code

Acting as a called program



```
L   R13,SAVE+4   OLD SAVE AREA
LM  R14,R12,12(R13) RESET REGS
LA  R15,0        RETURN CODE

BR  R14          BRANCH BACK
```

Caution!

- R1 is a volatile register
- R1 is can be modified by any IBM (or local) macro – OPEN, CLOSE, GET, PUT, …
- You must store off R1 before executing any of the above macros: LR R2,R1

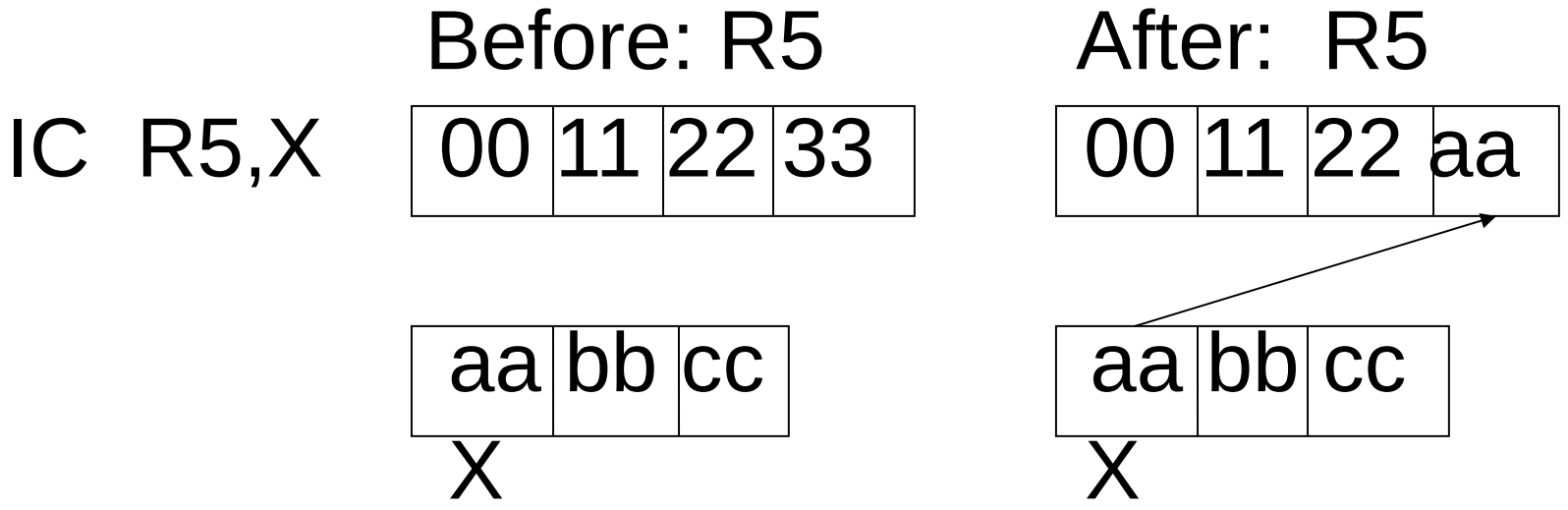
Exercise #6

- Create a subprogram that is passed 4 variables, A, B, C, and D. Compute (A*B)+C rounded to 2 decimals. Return the answer in D
- Assume each field is PL3 with 2 decimals
- Write a calling program that reads a file, passes four parameters to your subprogram and prints a report listing the value of each variable A, B, C, and the computed value D.

Working With Large Data Fields

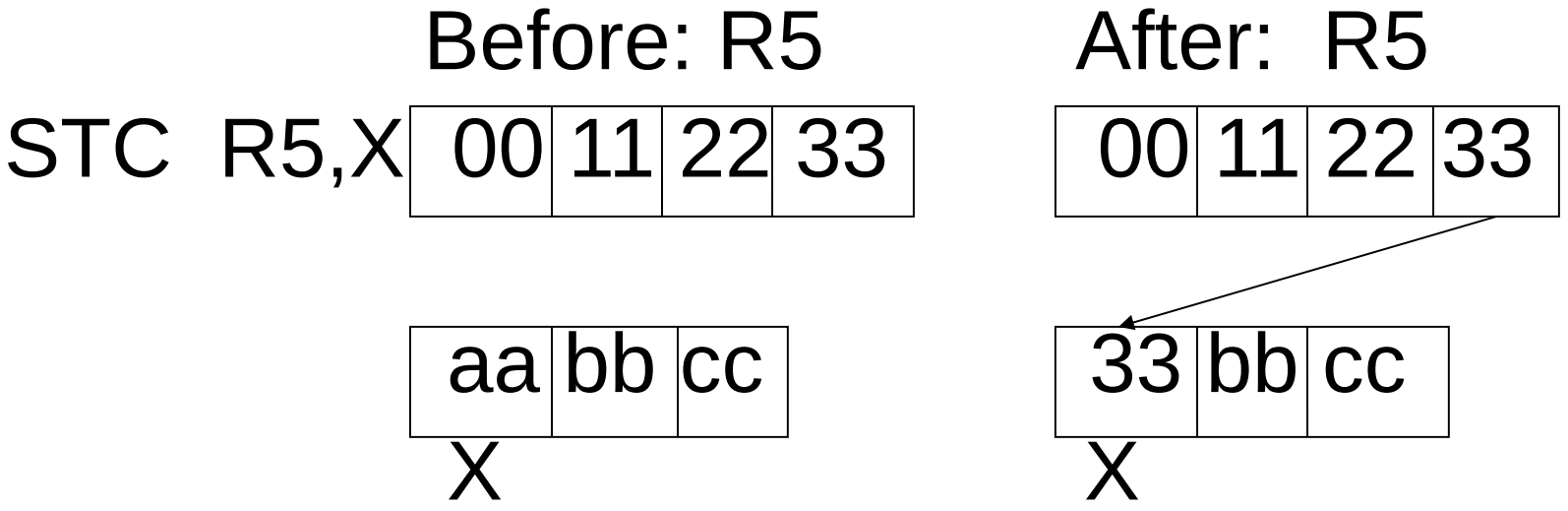
Insert Character

- RX
- Copies a single byte from memory (first byte) into the rightmost byte of a register



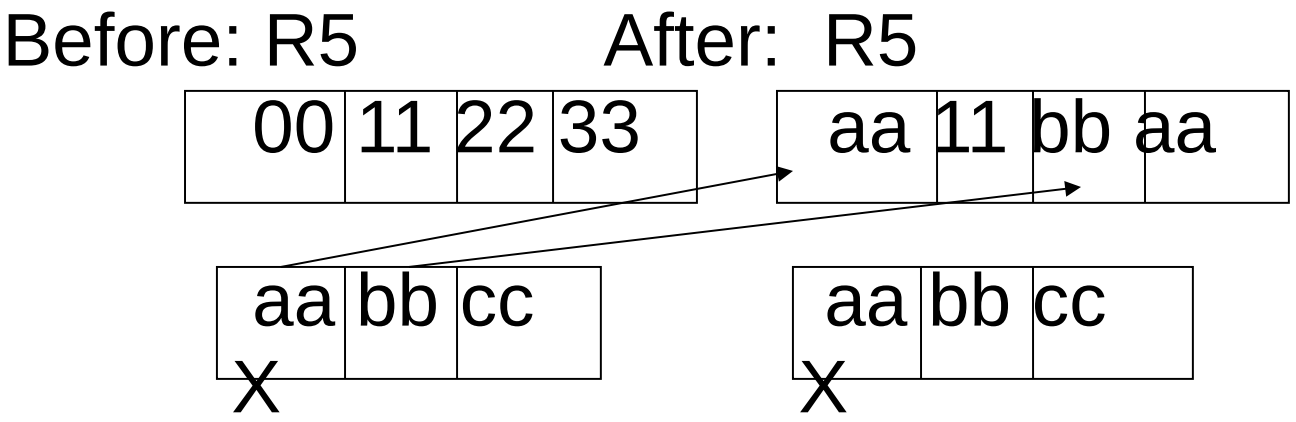
Store Character

- RX
- Copies a single byte from the rightmost byte of a register into memory



Insert Characters Under Mask

- RS
- Copies a consecutive bytes from memory (1-4 bytes) into the bytes of a register based on a binary mask.
- ICM R5,B’1010’,X



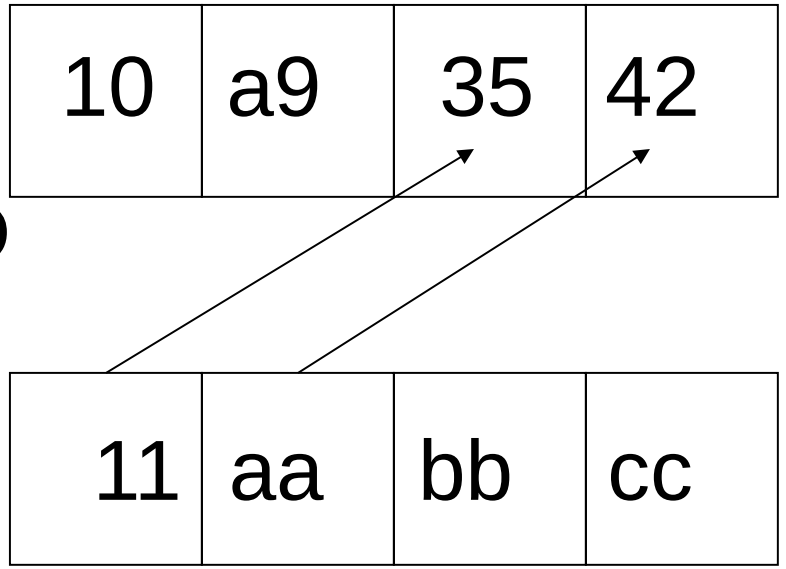
Insert Characters Under Mask

- ICM is sometimes used to load a word or halfword into a register from a memory location that isn't aligned on a halfword or fullword boundary

R7

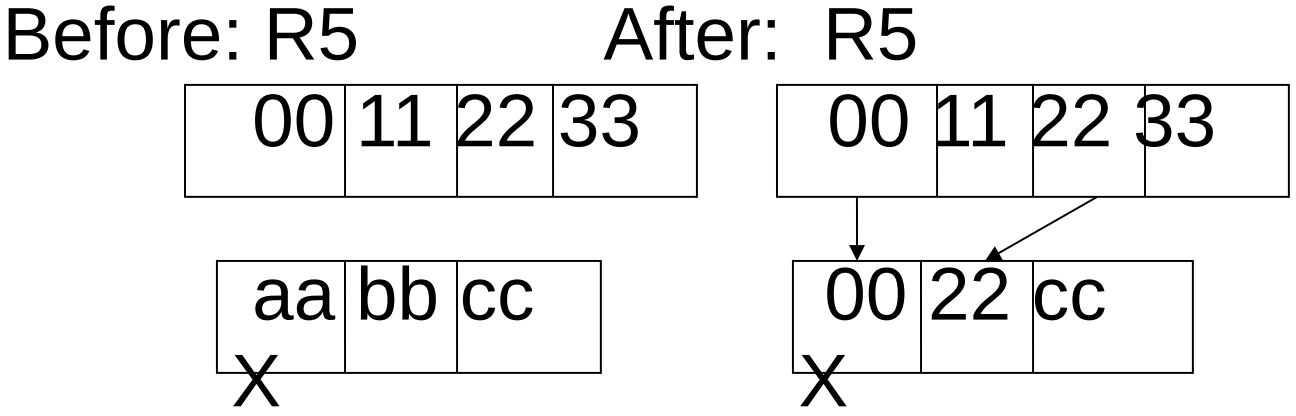
ICM R7,B'0011'XWORD

XWORD



Store Characters Under Mask

- RS
- Copies bytes of a register based on a binary mask into consecutive bytes of memory (1-4 bytes) .
- STCM R5,B’1010’,X



MVCL – Move Characters Long

- Used to move data in storage provided the source and target don’t overlap
- Uses four registers, two even/odd pairs
- Op 1 even register contains the target address
- Op 1 odd register contains the length of Op 1
- Op 2 even register contains the source address
- Op 2 odd register contains the length of Op 2
- Op 2 contains a pad byte in the first 8 bits

MVCL – Move Characters Long

Case 1: L1 > L2

Before execution:

R4	R5	R6	R7
A(A)	1000	A(B)	x’40’ 500

After execution:

R4	R5	R6	R7
A(A) + 1000	0	A(B) + 500	x’40’ 0

Padding occurs with 500 blanks (x’40’)

MVCL – Move Characters Long

Case 1: L1 < L2

Before execution:

R4	R5	R6	R7
A(A)	500	A(B)	x’40’ 1000

After execution:

R4	R5	R6	R7
A(A) + 500	0	A(B) + 500	x’40’ 500

No padding occurs

MVCL – Move Characters Long

Case 1: L1 = L2

Before execution:

R4	R5	R6	R7
A(A)	1000	A(B)	x’40’ 1000

After execution:

R4	R5	R6	R7
A(A) + 1000	0	A(B) + 1000	x’40’ 0

No padding occurs

MVCL – Move Characters Long

- MVCL does not permit sending and receiving fields to overlap
- MVCL sets the condition code:
  - CC = equal Fields equal in size
  - CC = low Size of field 1 < size of field 2
  - CC = high Size of field 1 > size of field 2
  - CC = overflow Fields overlapped
  - Test with BE, BL,BH, BO

MVCL Sample Code

```
 LA    R4,FIELDA   POINT AT TARGET FIELD WITH EVEN REG
 L     R5,LENGTHA  PUT LENGTH OF TARGET IN ODD REG
 LA    R6,FIELDB   POINT AT SOURCE FIELD WITH EVEN REG
 L     R7,LENGTHB  PUT LENGTH OF SOURCE IN ODD REG
 ICM   R7,B'1000',BLANK  INSERT BLANK PAD CHAR IN ODD REG
 MVCL   R4,R6
                                   …
FIELDA   DC    CL2000' '
BDATA    DC    1000CL1'X'
         ORG   BDATA
FIELDB   DS    CL1000
LENGTHA  DC    A(L'FIELDA)  CREATE ADDR CON AS  LENGTH
LENGTHB  DC    A(L'FIELDB)  CREATE ADDR CON AS A LENGTH
BLANK    DC    C' '
```

Blanking an Area with MVCL

```
LA    R8,TARGET
L     R9,TARLEN
LA    R4,SOURCE   SOURCE DOESN'T
PARTICIPATE
LA    R5,0        SET LENGTH OF SOURCE TO 0
ICM   R5,B'1000',BLANK  SET PAD TO A BLANK
MVCL  R8,R4               COPY BLANKS
```

CLCL - Compare Long

- Long fields can be compared using CLCL
- Just like MVCL, the setup involves two even/odd pairs for the source and target fields
- As long as the compared bytes are equal, CLCL adds 1 to the addresses in the even registers and decrements the odd registers by 1

CLCL - Compare Long

- Unequal bytes causes the operation to terminate with the address of the unequal bytes in the even registers and the condition code is set (equal, low, high, overflow)
- The pad character can be supplied for unequal sized fields and each pad character participates in the comparison

Using Multiple Base Registers

- An “ideal” program will have a single base register
- Few programs are “ideal”
- Many programs require multiple base registers
- Providing multiple base registers is a two step process
  1. The registers are declared in a USING
  2. Each register must be loaded with it’s base address

Using Multiple Base Registers

- There are as many ways to load base registers as there are programmers. Here's a simple approach to load 3 registers:

```
BASR  12,0
USING *,12,11,10
LA    R10,2048
LA    R11,2048(R10,R12)
LA    R10,2048(R10,R11)
```

Exercise #7

- Create a program that contains three 2000 byte fields. Define the fields like this:

```
FIELDA   0DS  CL2000
           DC  2000C'A'
FIELDB   0DS  CL2000
           DC  1000C'A'
           DC  1000C'B'
FIELDC     DC  CL2000' '
```

Exercise #7

- Move FieldB to FieldC.
- Print out FieldA, FieldB and FieldC in a series of 100 byte lines (use DSECTs)
- Compare FieldA to FieldC and print a message indicating which one is larger.
- You may find this helpful:

```
ALEN    DC   A(L'FIELDA)
BLEN    DC   A(L'FIELDB)
CLEN    DC   A(L'FIELDC)
```

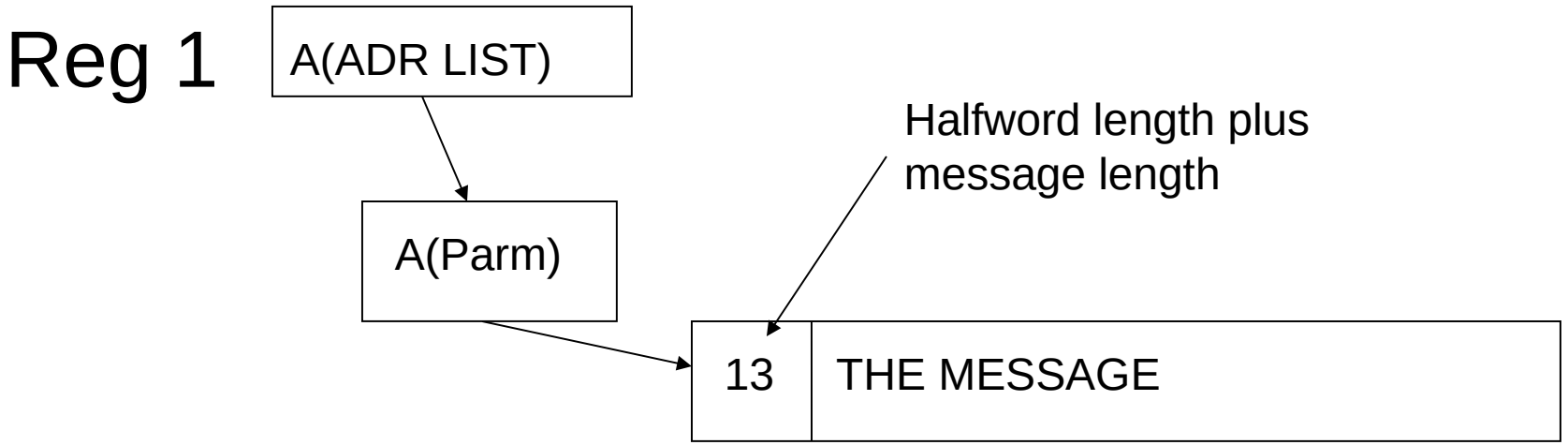
Working With Variable Length Data

Parm Data

- Here is some JCL that passes a parm

```
//COND00A EXEC PGM=JCLCONC1,
//         PARM='THE MESSAGE'
```

- Here is the data structure the parm creates



Processing the Parm Data

```
PARMSECT  DSECT
LEN       DS  H
PARMDATA  DS  CL256
```

```
     USING PARMSECT,R8
L     R8,0(R0,R1) R8 PTS AT PARM
LH    R9,LEN      GRAB THE PARM LEN
BCTR  R9,R0       SUB 1 FOR LENGTH
      EX R9,MOVE  MOVE THE DATA
      …
MOVE  MVC  PARMOUT(0),PARMDATA
```

Notes on Processing Parm Data

- The target MVC is coded like this

```
MOVE     MVC  PARMOUT(0),PARMDATA
```

- An explicit length of zero allows the “OR” operation of EX to “copy” a length into the target instruction temporarily
- Rightmost byte of R9 is “OR-ed” into the second byte of the target instruction (the length)
- EX R9,MOVE executes the MVC out of line

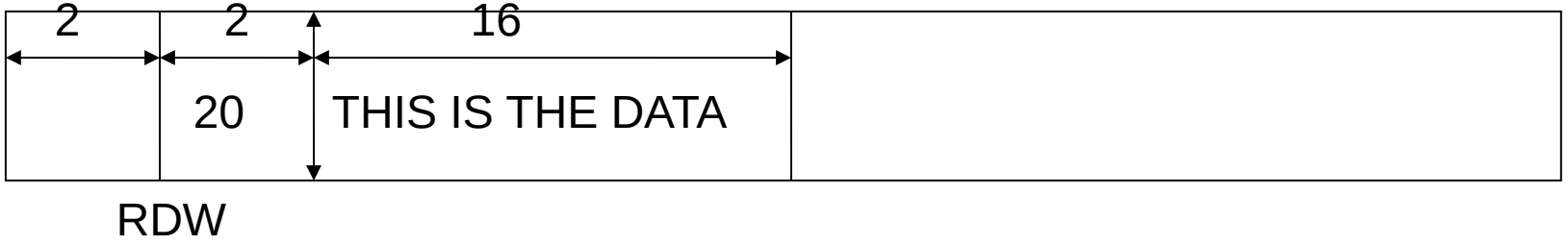
Exercise #8

- Write a program that prints the parm data it is passed through JCL
- Run the program three times with these JCL EXEC statements:

```
//COND00A EXEC PGM=JCLCONC1,PARM=‘THE'
//COND00B EXEC PGM=JCLCONC1,PARM=‘THE MESSAGE‘
//COND00C EXEC
  PGM=JCLCONC1,PARM=‘ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Variable Length Records

- Variable length records are described in the DCB with RECFM=VB or RECFM=V
- When the record is read, it arrives with a Record Descriptor Word (RDW) at the beginning.
- The RDW consists of two halfwords, the first is unused and the second contains the length of the record including the RDW



Reading VB or V Records

- The input buffer you define has to be large enough to accommodate the largest record + the RDW
- It could be defined like this:

```
            DS   0F     ALIGNMENT
MYREC       DS   0CL124
RDW         DS    0F
            DS     H
RECLEN      DS     H
DATA        DS     CL120
```

Reading VB or V Records

- After reading the record, load the RECLEN into a register
- Subtract 4 from the register to account for the RDW
- Subtract 1 from the register to account for an object code length
- Use EX to execute an MVC to an output area

Processing V or VB Records

```
          L    R8,RECLEN
          S    R8,=F'5'
          EX   R8,TARGET
          PUT  MYFILE,RECOUT
          …
TARGET    MVC  RECOUT(0),DATA
```

Writing V or VB Records

- Move the data to your output buffer. We can reuse MYREC.
- Determine the number of bytes in the logical record.
- Add four for the RDW. Store the record length + 4 in RECLEN
- PUT the record

Exercise #9

- Read the file BCST.SICCC01.PDSLIB(EXER9) which has records in the following format:
- Cols 1-2 Length in Character format
- Cols 3-80 Data
- Write a program with reads the file as 80 byte records and writes out a VB file using the length of each record to detemine how much data to write

Exercise #10

- Read the file VB file you produced in Exercise 9.
- Print each record using the length that is delivered in the RDW

Translate

- There is a special instruction for translating strings of characters called Translate (TR)
- Using TR you can easily convert a string or file from one format to another. For example ASCII to EBCDIC or lowercase letters to uppercase letters.
- One of the difficulties of using TR is that it requires you to build a table of values that indicate by there position how the translation will proceed

Translate

- In many cases you are interested in only changing a few of the values. For example, you may only want to change the 26 lowercase letters to uppercase.
- In these cases, there is an easy way to build the required table.

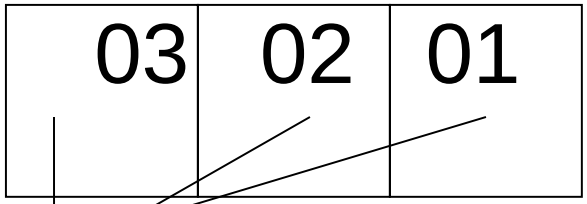
Translate

- SS1
- The first operand is the memory location that contains the data to be translated.
- The second operand is the translate table that tells how the translation will occur
- Each byte in the string we are translating is used as a displacement into the table. The corresponding table byte replaces the byte we are translating.
- Translation occurs from left to right in operand 1

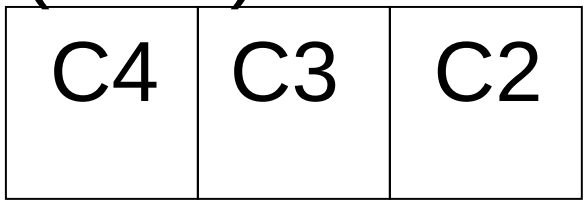
Translate

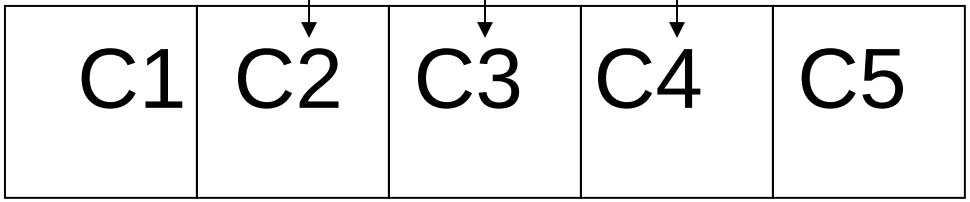
TR X,MYTABLE

X (Before)



X (After)





MYTABLE

Translate

- Since the string you are translating might contain any of 256 possible patterns, most TR tables have 256 bytes.
- Here is a table that translates every byte to itself:

  ```
  MYTABLE    DC   256AL1(*-MYTABLE)
  ```

- Starting with this as a basis, you can ORG back into the table to change the values you are really interested in.

Translate

- Here is a table that translates digits to blanks:

```
MYTABLE    DC  256AL1(* - MYTABLE)
                ORG  MYTABLE+C'0'
                DC   CL10' '
                ORG
```

Exercise #11

- Write a program that reads and prints a file called BCST.SICCC01.PDSLIB(EXER11) which contains records in lowercase letters (other characters, too).
- For each record, translate it to uppercase before printing the record.

Translate and Test

- TRT is somewhat misnamed as no translation occurs automatically.
- Instead, TRT is used to find character bytes that we are searching for. For example, TRT could be used to find a comma in a record, or the first blank.
- Like TR, TRT requires the programmer to build a TRT table

Translate and Test

- TRT tables are related to TR tables, but the semantics of the statement is different.
- Like TR the byte we are translating is used as a displacement into a table. If the table byte we find is X’00’, translation continues,otherwise translation terminates
- Finding any non-X’00’ byte stops the translation and test.

TRT

- TRT sets the condition code to indicate the results of the operation:
- (Zero) All function bytes encountered were X’00’.
- (Minus) A nonzero function byte was found before the end of operand 1
- (Positive) A nonzero function byte was found at the end of the operand 1

TRT Scan for $ or ?

```
TABLE    DC    256AL1(0)
         ORG TABLE+C'$'   Scan for $
         DC    X'FF'
         ORG   TABLE+C'?' Scan for ?
         DC    X'FF'
         ORG
         …
         TRT  MYSTRING,TABLE
         BZ   NOTFND
```

TRT Sets Regs 1 and 2

- If the TRT process finds a non X’00’ table byte (function byte), Reg 1 will be set with the address of the byte from the string that was used to find a non-zero function byte
- If the TRT process finds a non X’00’ table byte (function byte), Reg 2 will set the function byte into the rightmost byte of the register.
- Coding this will move the function byte to the string:

```
STC  R2,0(R0,R1)
```

Testing Numerics with TRT

```
TABLE       DC    256X'FF'
            ORG   TABLE+X'F0'
            DC    10X'00'   10 DIGITS OCCUR IN ORDER
            ORG
```

- Suppose we want to test a field called “FIELD” to see if it is numeric in the sense described above. This can be accomplished as follows.

```
            TRT   FIELD,TABLE
            BZ    ALLNUMS
            B     NOTNUMS
```

Exercise #12

- Read and print file BCST.SICCC01.PDSLIB(EXER12)
- Each record contains a last name, a comma, first name, #.
- Print each name as first name, space, last name.
- This will require working with variable length data and some address arithmetic in the registers.