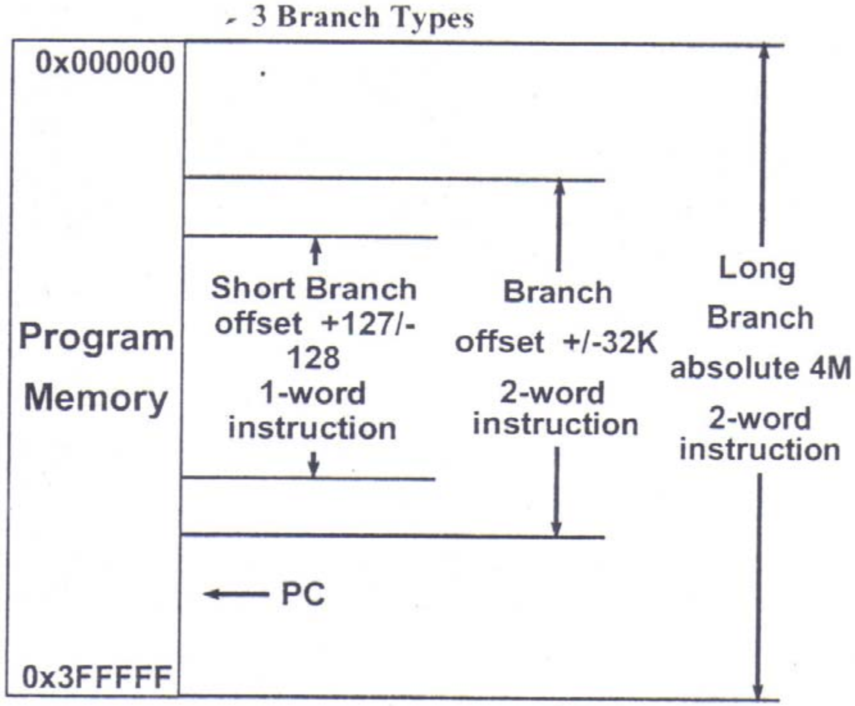
Branch Types and Range



*C28x DSP Design Workshop, Student guide, Rev. 6.2TTO, okt. 2005

Program Control - Branches

Function	Instruction		Cycles T/F	Size
Short Branch	SB	8bit,cond	7/4	1
Fast Short Branch	SBF	8bit,EQ|NEQ|TC|NTC	4/4	1
Fast Relative Branch	B	16bit,cond	7/4	2
Fast Branch	BF	16bit,cond	4/4	2
Absolute Branch	LB	22bit	4	2
Dynamic Branch	LB	*XAR7	4	1
Branch on AR	BANZ	16bit,ARn--	4/2	2
Branch on compare	BAR	16bit,ARn,ARn,EQ|NEQ	4/2	2

Condition Code

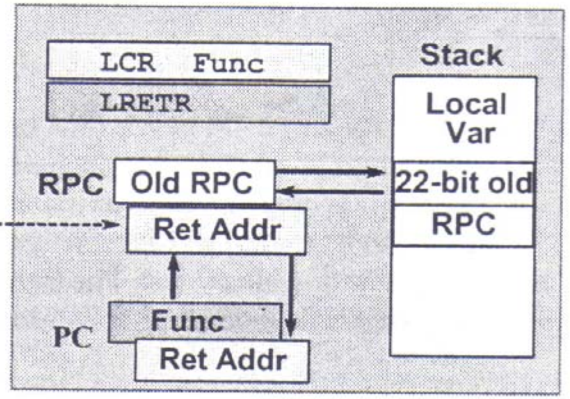
NEQ	LT	LO (NC)	NTC
EQ	LEQ	LOS	TC
GT	HI	NOV	UNC
GEQ	HIS (C)	OV	NBIO

- Condition flags are set on the prior use of the ALU
- The assembler will optimize B to SB if possible

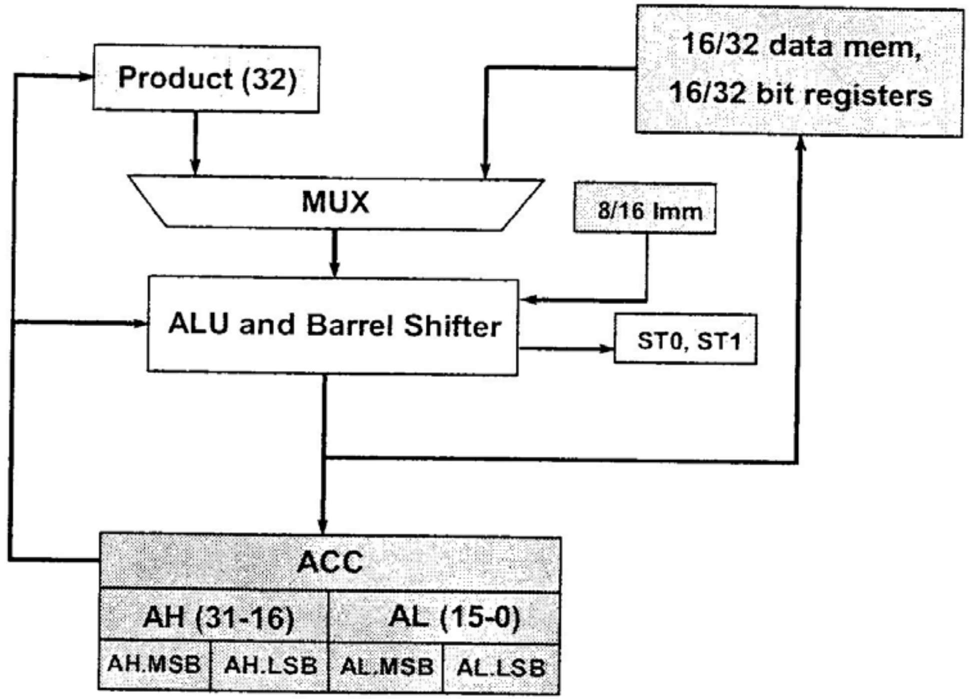
Program Control - Call/Return

Function	Call Code	Cycles	Return code	Cycles
Call	LCR 22bit	4	LRETR	4
Dynamic Call	LCR *XARn	4	LRETR	4
Interrupt Return			IRET	8

- More Call variations in the user guide are for code backward compatibility



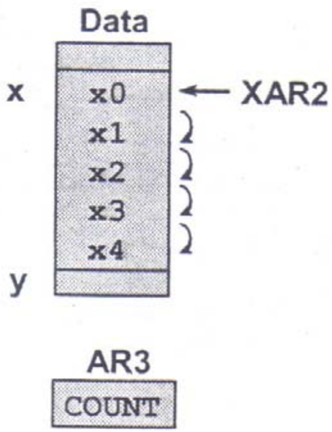
ALU and Accumulator



BANZ Loop Control Example

- Auxliary register used as loop counter
- Branch if Auxilary Register not zero
- Test performed on lower 16-bits of XARx only

$$y = \sum_{n=0}^{4} x_n$$




```
len     .set      5
x       .usect    "samp",6
y       .set      (x+len)

        .sect     "code"
        MOVL      XAR2,#x
        MOV       AR3,#len-2
        MOV       AL,*XAR2++
sum:    ADD       AL,*XAR2++
        BANZ      sum,AR3--
        MOV       *(0:y),AL
```

Accumulator - Basic Math Instructions

Format

```
xxx     Ax,  #16b   ;word
xxxB    Ax,  #8b    ;byte
xxxL    ACC, #32b   ;long
```

xxx = instruction: MOV, ADD, SUB, ...
Ax = AH, or AL
Assembler will automatically convert to 1 word instruction.

Ex

```
ADD     ACC, #01234h<<4
ADDB    AL,  #34h
```

Two word instructions with shift option
One word instruction, no shift

Variation

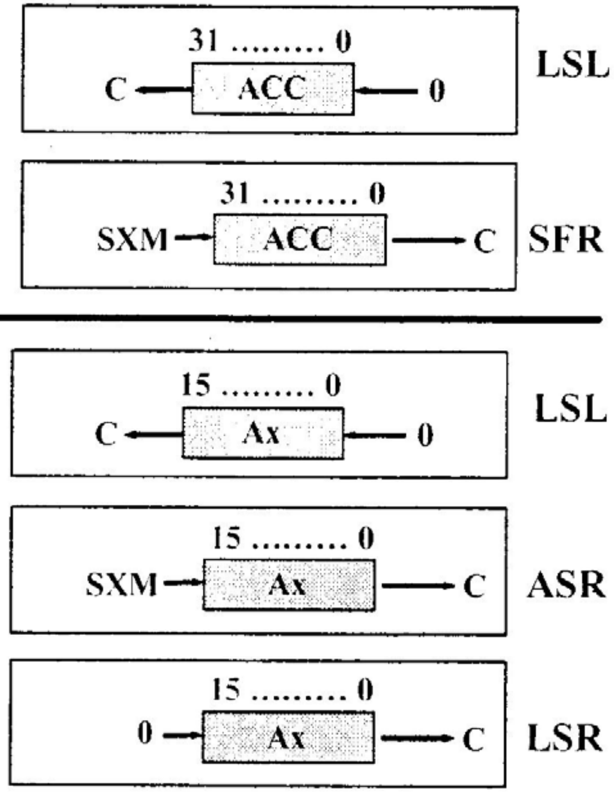
ACC Operations	
MOV ADD SUB	ACC,loc16<<shift from memory (left shift optional)
MOV ADD SUB	ACC,#16b<<shift 16-bit constant (left shift optional)
MOV loc16,ACC <<shift ;AL MOVH loc16,ACC <<shift ;AH	

Ax = AH or AL Operations	
MOV	Ax, loc16
ADD	Ax, loc16
SUB	Ax, loc16
AND	Ax, loc16
OR	Ax, loc16
XOR	Ax, loc16
AND	Ax,loc16,#16b
NOT	Ax
NEG	Ax
MOV	loc16,Ax

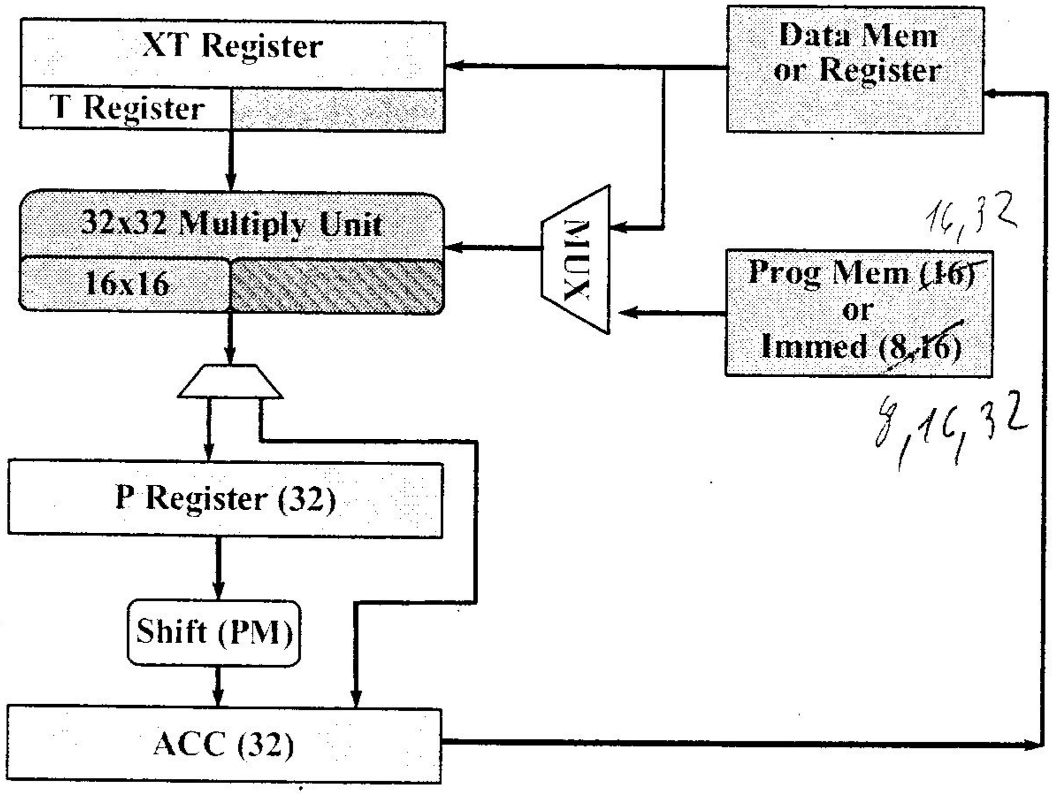
Shift the Accumulator

Shift full ACC		
LSL	ACC <<shift	(1-16)
SFR	ACC >>shift	
LSL	ACC <<T	(0-15)
SFR	ACC >>T	

Shift AL or AH	
LSL	AX <<shift
LSR	AX <<shift
ASR	AX >>shift
LSL	AX <<T
LSR	AX <<T
ASR	AX >>T



Multiply Unit



Multiplier Instructions

Instruction	Execution	Purpose
MOV T,loc16	T = loc16	Get first operand
MPY ACC,T,loc16	ACC = T*loc16	For single or first product
MPY P,T,loc16	P = T*loc16	For n^{th} product
MPYB ACC,T,#8bu	ACC = T*8bu	Using 8-bit unsigned const
MPYB P,T,#8bu	P = T*8bu	Using 8-bit unsigned const
MOV ACC,P	ACC = P	Move 1^{st} product<<PM to ACC
ADD ACC,P	ACC += P	Add n^{th} product<<PM to ACC
SUB ACC,P	ACC -= P	Sub n^{th} product<<PM fr. ACC

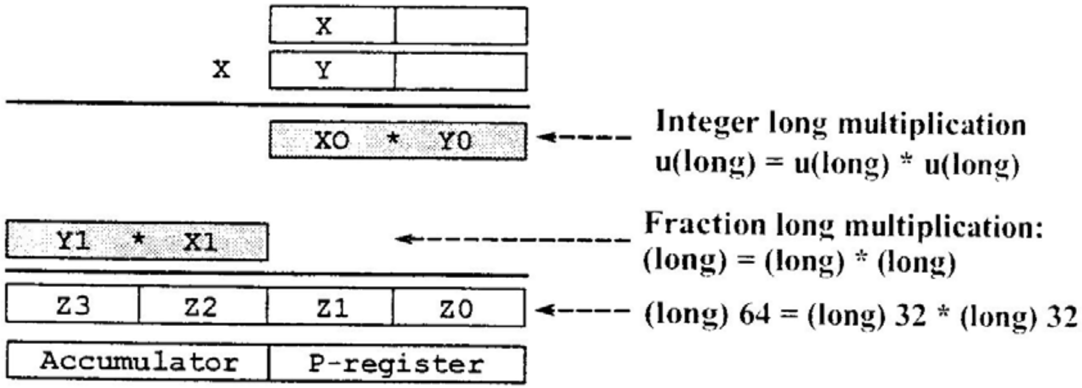
Instruction	Execution	
MOVP T, loc16	ACC = P<<PM	T = loc16
MOVA T, loc16	ACC += P<<PM	T = loc16
MOVS T, loc16	ACC - = P<<PM	T = loc16
MPYA P, T, #16b	ACC += P<<PM	*then* P = T*#16b
MPYA P, T, loc16	ACC += P<<PM	*then* P = T*loc16
MPYS P, T, loc16	ACC - = P<<PM	*then* P = T*loc16

Sum-of-Products

Y = A*X1 + B*X2 + C*X3 + D*X4

```
ZAPA                 ;ACC = P = OVC = 0
MOV   T,@X1          ;T = X1
MPY   P,T,@A         ;P = A*X1
MOVA  T,@X2          ;T = X2   ;ACC = A*X1
MPY   P,T,@B         ;P = B*X2
MOVA  T,@X3          ;T = X3 ;ACC = A*X1 + B*X2
MPY   P,T,@C         ;P = C*X3
MOVA  T,@X4          ;T = X4;ACC = A*X1 + B*X2 + C*X3
MPY   P,T,@D         ;P = D*X4
ADDL  ACC,P<<PM      ;ACC = Y
MOVL  @y,ACC
```

32x32 Long Multiplication



IMPYAL	P,XT,loc32	P = u(XT)*u(loc32)
QMPYAL	ACC,XT,loc32	ACC = (XT)*(loc32)

IMACL	P,loc32,*XAR7	ACC += P; P = u(loc32)*u(loc32)
QMACL	P,loc32,*XAR7	ACC += P; P = (loc32)*(loc32)

Repeat Next: RPT

- Options:

RPT #8bit	up to 256 iterations
RPT loc16	location "loc16" holds count value

- Features:
  - Next instruction iterated N+1 times
  - Saves code space - 1 word
  - Low overhead - 1 cycle
  - Easy to use
  - Non-interruptible
  - Requires use of || before next line
  - May be nested within BANZ loops

Example :

```
int x[5]={0,0,0,0,0};
```

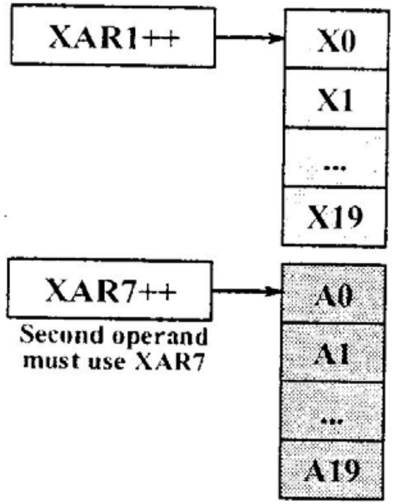
```
x     .usect "samp",5
      MOV    AR1,#x
      RPT    #4
||    MOV    *XAR1++,#0
```

Instruction	Cycles
RPT	1
BANZ	4·N

Refer to User Guide for more repeatable instructions

Sum-of-Products: RPT / MAC

$$y=\sum_{n=0}^{19} x_n a_n$$




```
x      .usect  "sample",20
y      .usect  "result",2
       .sect   "coefficient"
a0:    .word   0x0101
       .word   0x0202
          • • •
       .word   0x2020
       .sect   "code"
SOP:   SPM     0
       MOVW    DP,#y
       MOVL    XAR1,#x
       MOVL    XAR7,#a0
       ZAPA                      <- Zero ACC & P
       RPT     #19               <- Repeat single
||     MAC     P,*XAR1++,*XAR7++ <- Dual operand
       ADDL    ACC,P<<PM         <- last ADD
       MOVL    @y,ACC
       B       SOP,UNC
```

```
MOV   T,loc16  }    MOVA  T,loc16    }        { ACC+=P
ADD   ACC,P    }    MPY   P,T,loc16  }  MAC   { T=*ARn++
                                              { P=T*(*ARn++)
```

Data Move Instructions

DATA ↔ DATA (4G ↔ 64K)	DATA ↔ PGM (4G ↔ 4M)
MOV loc16, *(0:16bit)	PREAD loc16 ,*XAR7
MOV *(0:16bit), loc16	PWRITE *XAR7, loc16

16-bit address concatenated with 16 leading zeros

32-bit address memory location

pointer with a 22-bit program memory address

```
        .sect   ".code"
START:  MOVL    XAR5,#x
        MOVL    XAR7,#TBL
        RPT     #len-1
||      PREAD   *XAR5++,*XAR7
        ...
x       .usect  ".samp",4
        .sect   ".coeff"
TBL:    .word   1,2,3,4
len     .set    $-TBL
```

- Optimal with RPT (speed and code size)
- In RPT, non-mem address is auto-incremented in PC
- Faster than Load / Store, avoids accumulator
- Allows access to program memory

Conditional Moves

Instruction	Execution (if COND is met)
`MOV loc16,AX,COND`	`[loc16] = AX`
`MOVB loc16,#8bit,COND`	`[loc16] = 8bit`

Instruction	Execution (if COND is met)
`MOVL loc32,ACC,COND`	`[loc32] = AX`

Example

If A<B, Then B=A

```
A   .usect    "var",2,1
B   .set      A+1
    .sect     "code"
    MOVW  DP, #A
    MOV   AL, @A
    CMP   AL, @B
    MOV   @B,  AL, LT
```

Accumulator

0 0 0 0 | 0 1 2 0

Data Memory (Before)		Data Memory (After)	
0 1 2 0	A	0 1 2 0	A
0 3 2 0	B	0 1 2 0	B

Before | After

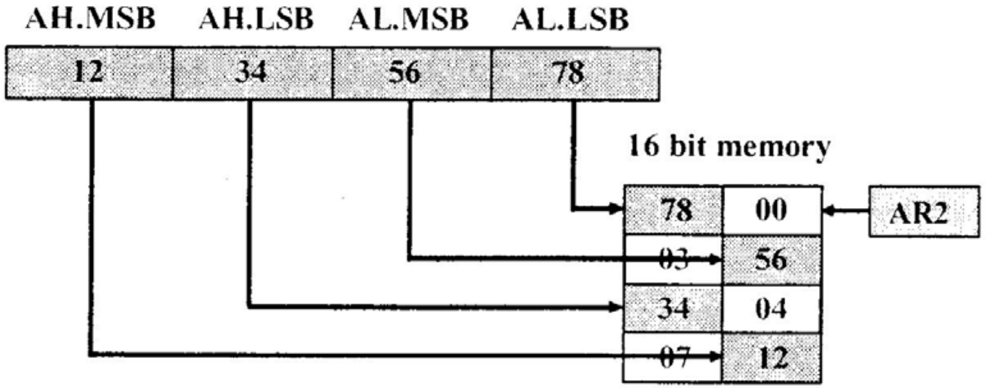
Byte Operations

MOVB AX.LSB,loc16	0000 0000	Byte	AX
MOVB AX.MSB,loc16	Byte	No change	AX
MOVB loc16, AX.LSB	No change	Byte	loc16
MOVB loc16, AX.MSB	No change	Byte	loc16

Byte = 1. Low byte for register addressing
2. Low byte for direct addressing
3. *Selected* byte for offset indirect addressing

For loc16 = *+XARn[Offset]	Odd Offset	Even Offset	loc16

Byte Addressing



Example of Byte Un-Packing

```
MOVL  XAR2, #MemA
MOVB  *+XAR2[1], AL.LSB
MOVB  *+XAR2[2], AL.MSB
MOVB  *+XAR2[5], AH.LSB
MOVB  *+XAR2[6], AH.MSB
```

Example of Byte Packing

```
MOVL  XAR2, #MemA
MOVB  AL.LSB,*+XAR2[1]
MOVB  AL.MSB,*+XAR2[2]
MOVB  AH.LSB,*+XAR2[4]
MOVB  AH.MSB,*+XAR2[7]
```

Test and Change Memory

Instruction		Execution	Affects
TBIT	loc16,#(0-15)	ST0(TC) = loc16(bit_no)	TC
TSET	loc16,#(0-15)	Test (loc16(bit)) then set bit	TC
TCLR	loc16,#(0-15)	Test (loc16(bit)) then clr bit	TC
CMPB	AX, #8bit	Test (AX - 8bit unsigned)	C,N,Z
CMP	AX, loc16	Test (AX – loc16)	C,N,Z
CMP	loc16,#16b	Test (loc16 - #16bit signed)	C,N,Z
CMPL	ACC, @P	Test (ACC - P << PM)	C,N,Z

MIN/MAX Operations

Instruction	Execution
MAX ACC,loc16	if ACC < loc16, ACC = loc16 if ACC >= loc16, do nothing
MIN ACC,loc16	if ACC > loc16, ACC = loc16 if ACC <= loc16, do nothing
MAXL ACC,loc32	if ACC < loc32, ACC = loc32 if ACC >= loc32, do nothing
MINL ACC,loc32	if ACC > loc32, ACC = loc32 if ACC <= loc32, do nothing
MAXCUL P,loc32 (for 64 bit math)	if P < loc32, P = loc32 if P >= loc32, do nothing
MINCUL P,loc32 (for 64 bit math)	if P > loc32, P = loc32 if P <= loc32, do nothing

Find the maximum 32-bit number in a table:

```
      MOVL   ACC,#0
      MOVL   XAR1,#table
      RPT    #(table_length – 1)
||    MAXL   ACC,*XAR1++
```

Read-Modify-Write Instructions

- Work directly on memory – bypass ACC
- Atomic Operations – protected from interrupts

AND	loc16,AX
OR	loc16,AX
XOR	loc16,AX
ADD	loc16,AX
SUB	loc16,AX
SUBR	loc16,AX
INC	loc16
DEC	loc16

AH, AL

AND	loc16,#16b
OR	loc16,#16b
XOR	loc16,#16b
ADD	loc16,#16b
SUBR	loc16,#16b
TSET	loc16,#bit
TCLR	loc16,#bit

16- bit constant

Read-Modify-Write Examples

update with a mem	update with a constant	update by 1
VarA += VarB	VarA += 100	VarA += 1

SETC INTM
MOV AL, @VarB
ADD AL, @VarA
MOV @VarA, AL
CLRC INTM
	SETC INTM
	MOV AL, @VarA
	ADD AL, #100
	MOV @VarA, AL
	CLRC INTM
		SETC INTM
		MOV AL, @VarA
		ADD AL, #1
		MOV @VarA, AL
		CLRC INTM

MOV AL, @VarB ADD @VarA, AL	ADD @VarA,#100	INC @VarA

$\underbrace{\hspace{10cm}}$

Benefits of Read-Modify-Write Instructions