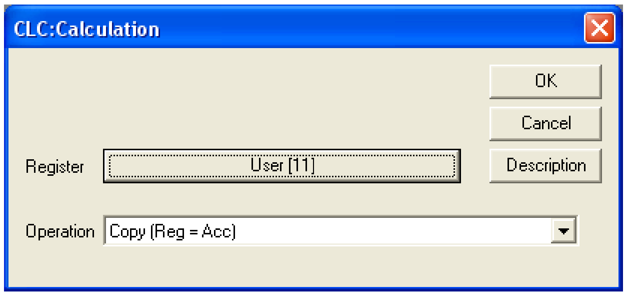
Calculation Commands

Basic math and logic functions can be a vital component to programs written in almost any language. The SilverLode servos use the Calculation (CLC), the Calculation Extended (CLX), and the Calculation Data (CLD) commands for these functions.

The CLC command uses only two words of memory and is used for basic math operations (see below). CLC is supported by all SilverLode servos. The CLX and CLD commands allow for three register operations (see below). CLX and CLD are supported by the SilverDust only.

CLC:Calculation



Register: User [11]
Operation: Copy (Reg = Acc)

Overview

The parameters to the CLC command include:
Register
Operation

The Operations, such as Add, Sub, Mult, and DIv are described below. In each description the use of Register parameter is detailed.

Accumulator (Register 10)

SilverLode commands are closely related to assembly language. An important similarity to assembly for the CLC command is the use of an Accumulator register (register 10) for calculations.

Two operand operations like,

User[25] = User[26] x User[27],

require multiple CLC commands as follows:

Copy (Accumulator[10] = User[26])
Mult (Accumulator[10] = Accumulator[10] * User[27])
Save (User[25] = Accumulator[10])

The first command copies the value of register 26 into the Accumulator, the second command multiplies the value in the Accumulator with the value in register 27 and places the result in the Accumulator, and the third command stores the value of the Accumulator in register 25.

QuickControl® is a registered trademark of QuickSilver Controls, Inc.
Other trade names cited are property of their explicit owner.

Operations

Absolute Value (Reg = Abs(Reg)): This command replaces the value in the selected register with a positive value of the same magnitude.

Example:	User[11] = -64	
	Absolute Value (Reg = ABS(Reg))	*User[11] = Abs(User[11])*
Result:	User[11] = 64	

Add (Acc = Acc + Reg): This command adds the value in Accumulator[10] to the value in the selected register and stores the result in the Accumulator[10]. Note that three commands are needed to add the values from two different registers (see Example 2).

Example 1:	Accumulator[10] = 0	
	User[11] = 25	
	Add (Acc = Acc + Reg)	*Accumulator[10] = Accumulator[10] + User[11]*
Result:	Accumulator[10] = 25	

Example 2:	User[26] = 10	
	Copy (User[25])	*Accumulator[10] = User[25]*
	Add (User[26])	*Accumulator[10] = Accumulator[10] + User[26]*
	Save (User[27])	*User[27] = Accumulator[10]*
Result:	User[27] = User[25] + User[26]	

Add (Acc = HI(Reg) + Acc): This command adds the value in Accumulator[10] to the value in the upper word of a 32-bit register and stores the result in the Accumulator.

Example:	Accumulator[10] = 0	
	Add (Acc = HI(Reg) + Acc)	*Accumulator[10] = HI(User[11]) + Accumulator[10]*
	User[11] = 0x0001 0000 (65536)	
	Accumulator[10] = 0x0000 0100 (256)	
Result:	Accumulator[10] = 0x0000 0101 (257)	

Accumulator 10 now contains 257

Add (Acc = LO(Reg) + Acc): This command adds the value in Accumulator[10] to the value in the lower word of a 32-bit register and stores the result in the Accumulator.

Example: Accumulator[10] = 0
Add (Acc = LO(Reg) + Acc) *Accumulator[10] = HI(User[11]) + Accumulator[10]*
User[11] = 0x1000 0001
Accumulator[10] = 0x0000 0001
Result: Accumulator[10] = 0x0000 0002

Bitwise AND (Acc = Acc AND Reg): This command performs a bitwise "AND" on the Accumulator[10] value with the selected register value. The result is placed in Accumulator[10]. This means that the command compares each bit of both values and if both bits equal 1 or HIGH, the command places a 1 or HIGH in the result bit. Any other combination places a 0 or LOW. The best example of this operation is the binary display. (Note: all 32 bits are evaluated.)

Example: User[11] = 00101011 11001101, 0x0000 2BCD
Accumulator[10] = 01011100 10111010, 0x0000 5CBA
AND (Acc = Acc AND Reg) *Accumulator[10] = Accumulator[10] AND User[11]*
Result: Accumulator[10] = 00001000 10001000, 0x0888

Clear (Reg = 0): This command sets the selected register equal to zero. The CLX and CLD command do not use this function.

Example: User[11] = 68
Clear (Reg = 0) *User[11] = 0*
Result: User[11] = 0

Copy (Acc = Reg): This command copies the value of a selected register to Accumulator[10]. This command loads data to the accumulator for future operations (see **Add** and **Mult**).

Example: Copy(Acc = Reg) *Accumulator[10] = User[27]*
Result: Accumulator[10] now contains the value stored in User[27]

Copy Reg Ref (Acc = reg#, reg# = Value of Reg): The full name of this command is Copy from Register Reference. The register reference is similar to a pointer in C. This command retrieves the referenced value and stores in the selected register. This value is subsequently copied to Accumulator[10].

Example: User[27] = 100
User[11] = 27
Copy Reg Ref (Acc = reg#, reg# = Value of Reg)
Accumulator[10] = User[User[11]]
Result: Accumulator[10] = 100

Copy Reg Ref (reg# = Acc, reg# = Value of Reg): The full name of this command is Copy from Register Reference. The register reference is similar to a pointer in C. This command retrieves the referenced value and stores in the selected register. This value is subsequently copied to the register number corresponding to the value in the selected register.

```
Example:   Accumulator[10] = 100
           User[27] = 0
           User[11] = 27
           Copy Reg Ref (reg# = Acc, reg# = Value of Reg)   User[27] = Accumulator[10]
Result:    User[27] = 100
```

Copy Word (HI(Acc) = LO(Reg)): This command will copy the low word (bits 0-15) of the selected register to the high word (bits 16-32) of the Accumulator[10]. The best example of this operation is the binary display.

```
Example:   Accumulator[10] = 00000000 00000000 00000000 00000000, 0x0
           User[11]        = 00000000 00000000 00000000 11010011, 0xD3
           Copy Word (HI(Acc) = LO(Reg))         HI(Accumulator[10]) = LO(User[11])
Result:    Accumulator[10]  = 00000000 11010011 00000000 00000000, 0xD30000
```

Copy Word (HI(Reg) = LO(Acc)): This command will copy the low word (bits 0-15) of the Accumulator[10] to the high word (bits 16-32) of the selected register. The best example of this operation is the binary display.

```
Example:   User[11]        = 00000000 00000000 00000000 00000000, 0x0
           Accumulator[10] = 00000000 00000000 11001000 11010011, 0xC8D3
           Copy Word (HI(Reg) = LO(Acc))         HI(User[11]) = LO(Accumulator[10])
Result:    User[11]        = 11001000 11010011 00000000 00000000, 0xC8D30000
```

Copy Word (LO(Reg) = LO(Acc)): This command will copy the low word (bits 0-15) of the Accumulator[10] to the low word (bits 0-15) of the selected register. The best example of this operation is the binary display.

```
Example:   User[11]        = 00000000 00000000 00000000 00000000, 0x0
           Accumulator[10] = 00000000 00000000 11001000 11010011, 0xC8D3
           Copy Word (LO(Reg) = LO(Acc))      HI(User[11]) = LO(Accumulator[10])
 Result:   User[11]        = 00000000 00000000 11001000 11010011, 0xC8D3
```

Copy Word, Sign Extend (Acc = HI(Reg)): This command will copy the high word (bits 16-32) of the selected 32-bit register to the low word (bits 0-15) of the Accumulator[10]. If the MSB of the register was a 1, then the upper half of the accumulator is filled with 1's to sign extend. This allows a 16 bit signed number to be converted to a 32 bit signed number. The best example of this operation is the binary display.

Example: User[11] = 10101011 11001101 10001001 00111111, 0xABCD893F
Accumulator[10] = 00000000 00000000 00000000 00000000, 0x0
Copy Word, Sign Extend (Acc = HI(Reg))
Accumulator[10] Low Word = User[11] High Word
Result: Accumulator[10] = 11111111 11111111 10101011 11001101, 0xFFFF ABCD

Copy Word, Sign Extend (Acc = LO(Reg)): This command will copy the low word (bits 0-15) of the selected 32-bit register to the low word (bits 0-15) of the Accumulator[10]. This allows a 16 bit signed number to be converted to a 32 bit signed number. The best example of this operation is the binary display.

Example: User[11] = 10101011 11001101 10001001 00111111, 0xABCD893F
Accumulator[10] = 00000000 00000000 00000000 00000000, 0x0
Copy Word, Sign Extend (Acc = LO(Reg))
Accumulator[10] Low Word = User[11] High Word
Result: Accumulator[10] = 11111111 11111111 10001001 00111111, 0x FFFF 893F

Decrement (Reg = Reg – 1): This command will decrement the selected register by 1.

Example: User[11] = 100
Decrement (Reg = Reg - 1) *User[11] = User[11] -1*
Result: User[11] = 99

Div – Signed 32/16 bit (Acc = Acc / LO(Reg)): This command divides the value stored in Accumulator[10] by the low word (bits 0-15) of the selected register value and stores a signed result in the Accumulator[10]. The initial value in the Accumulator can be a signed 32-bit integer, but if the value in the selected register is not a 16-bit integer, it ignores the upper word (bits 16-31). Note that this is not a floating-point calculation. The result is always an integer and any remainder is lost. (The remainder may be found using the Calc Modulo as a separate calculation.)

Value Ranges:
Accumulator[10] = -2,147,483,648 to 2147483647 (or $-2^{(31)}$ to $2^{(31)} - 1$)
Selected Register = 0 - 65535 (or 0 to $2^{16} - 1$)

Example 1: Accumulator[10] = 100
User[11] = 10
Div – Signed 32/16 bit (Acc = Acc / LO(Reg))
Accumulator[10] = Accumulator[10] / LO(User[11])
Result: Accumulator[10] = 10

Increment (Reg = Reg + 1): This command will increment the selected register by 1.

Example: User[11] = 100
Increment (Reg = Reg + 1) *User[11] = User[11] +1*
Result: User[11] = 101

Max (Acc = Max(Acc, Reg)): This command will compare Accumulator[10] with the selected register. Accumulator[10] will then equal the greater value.

```
Example:   Accumulator[10] = 50
           User[11] =        10
           Max (Acc = Max(Acc, Reg))          Accumulator[10] > User[11]
Result:    Accumulator[10] = 50
```

Min (Acc = Min(Acc, Reg)): This command will compare Accumulator[10] with the selected register. Accumulator[10] will then equal the lesser value.

```
Example:   Accumulator[10] = 50
           User[11] =        10
           Max (Acc = Max(Acc, Reg))          Accumulator[10] > User[11]
Result:    Accumulator[10] = 10
```

Modulo 32 % 16 Bit (Acc % LO(Reg)): This command divides the value stored in Accumulator[10] by the lower word (bits 0-15) of the value stored in the selected register and stores the remainder of the division in Accumulator[10]. The command ignores the upper 16 bits (bits 16-31) of the value in the selected register.

```
Example:   Accumulator[10] = 33
           User[11] = 10
           Modulo 32 % 16 Bit (Acc % LO(Reg))
                                  Accumulator[10] = Accumulator[10] % User[11]
Result:    Accumulator[10] = 3
```

Mult - Signed (Acc = Acc * Reg): This command multiplies the signed Accumulator[10] with the signed selected register. Accumulator[10] stores the signed result.

```
Example:   Accumulator[10] = 5
           User[11] =        -10
           Mult – Signed (Acc = Acc * Reg)  Accumulator[10] = Accumulator[10] * User[11]
Result:    Accumulator[10] = -50
```

Mult - Unsigned (Acc = Acc * Reg): This command multiplies the unsigned Accumulator[10] with the unsigned selected register. Accumulator[10] stores the unsigned result.

```
Example:   Accumulator[10] = 5
           User[11] =        10
           Mult(Acc * Reg)              Accumulator[10] = Accumulator[10] * User[11]
Result:    Accumulator[10] = 50
```

Negative (Reg = -Reg): This command will multiply the selected register by –1.

```
Example:   User[11] = 100
           Negative(-Reg)               User[11] = -User[11]
Result:    User[11] = -100
```

Bitwise OR (Acc = Acc OR Reg): This command performs a bitwise “OR” on the Accumulator[10] value with the selected register value. The result is placed in Accumulator[10]. This means that the command compares each bit of both values and if either or both bits equal 1 or HIGH, the command places a 1 or HIGH in the result bit. Only if both bits equal 0, the result bit will be a 0 or LOW. The best example of this operation is the binary display.

Example: User[11] = 00000000 00000000 10001011 11001101, 0x8BCD
Accumulator[10] = 00000000 00000000 11011100 10111010, 0xDCBA
Bitwise OR (Acc = Acc OR Reg) *Accumulator[10] = Accumulator[10] OR User[11]*
Result: Accumulator[10] = 00000000 00000000 11011111 11111111, 0xDFFF

Shift Reg Left: This command performs a binary bit shift left operation on the selected register. This means that each bit of the value in the register moves to the next higher bit, effectively multiplying the original value by 2. The lowest bit is always set to 0 after the operation. The result of the operation is placed in the selected register, replacing the original value. This operation only has meaning when the value of the register is viewed as a binary number.

Example: User[11] = 00100100 10101101 11010110 11010101, 0x24ADD6D5
Shift Reg Left
Result: User[11] = 01001001 01011011 10101101 10101010, 0x495BADAA

Shift Reg Right w/ Sign Extend: This command performs a binary bit shift right operation on the selected register. This means that each bit of the value in the register moves to the next lower bit, effectively dividing the original value by 2. The highest bit (bit 31) is not changed in order to preserve the original sign (1 is negative, 0 is positive), although a 1 in bit 31 is shifted to bit 30. The lowest bit is discarded. The result of the operation is placed in the selected register, replacing the original value. This provides a divide by 2 on a 32 bit signed number.

Example: User[11] = 10100100 10101101 11010110 11010101, 0xA4ADD6D5
Shift Reg Right w/ Sign Extend
Result: User[11] = 11010010 01010110 11101011 01101010, 0xD256EB6A

Shift Reg Right w/o Sign Extend: This command is identical to **Shift Reg Right w/ Sign Extend** except that a 0 is placed in bit 31 of the selected register. This performs a divide by 2 on an unsigned 32 bit number.

Example: User[11] = 10100100 10101101 11010110 11010101, 0xA4ADD6D5
Shift Reg Right w/o Sign Extend
Result: User[11] = 01010010 01010110 11101011 01101010, 0x5256EB6A

Sub (Acc = Acc – Reg): This command subtracts the value in the selected register from the value in Accumulator[10] and stores the result in Accumulator[10]. Note that three commands are needed to subtract a value from another selected regsiter (see Example 2).

Example 1:	Accumulator[10] = 10	
	User[11] = 25	
	Sub(Acc = Acc - Reg)	*Accumulator[10] = Accumulator[10] - User[11]*
Result:	Accumulator[10] = -15	

Example 2:	User[26]= 10	
	Copy (User[25])	*Accumulator[10] = User[25]*
	Add (User[26])	*Accumulator[10] = Accumulator[10] + User[26]*
	Save (User[27])	*User[27] = Accumulator[10]*
Result:	User[27] = User[25] - User[26]	

Sub (Acc = Reg – Acc): This command subtracts the value in Accumulator[10] from the value in the selected register and stores the result in Accumulator[10].

Example 1:	Accumulator[10] = 10	
	User[11] = 25	
	Sub (Acc = Reg - Acc)	*Accumulator[10] = User[11] - Accumulator[10]*
Result:	Accumulator[10] = 15	

Sub Target Position (Targ – Reg, Pos – Reg): This command subtracts the value in the selected register from both Target Position[0] and Actual Position[1]. If the value in both registers were be modified separately using the **Sub** command and the Accumulator, the sudden change in target versus position (when only the first had been modified) would cause the motor to suddenly jump due to the control loop seeing a large difference, and the error limits and kill motor could quite likely be triggered. This command provides a simple way to adjust the effective zero point of the system to the value in Reg. This command is especially useful in the homing routines; the value of the last index is stored in Register 2, and the value of the last position which triggered a motion stop is stored in Register 4. Using this command using Register 2 will cause the last index location to become the zero point for the system, while using Register 4 will cause the location at which the last sensor (motion stop condition) position to become the zero point for the system. The final position following the move is not accurate as the motion must slow down after the sensor is found. Note: Other registers may be used to offset the zero based on a predetermined alignment value, for example.

Example: If a homing motion were made, using stop conditions based on IO3 going low, and the sensor was found at location 4400 :

Target Position[0] = 4510
Actual Position[1] = 4500
Last Trig Position[4] = 4400
Sub Target Position (Targ – Reg, Pos – Reg)
Target Position[0] = Target Position[0] - Last Trig Position[4]
Actual Position[1] = Actual Position[1] - Last Trig Position[4]

Result: Target Position[0] = 110
Actual Position[1] = 100

Performing an absolute move to location 0 would bring the motor to the location where IO3 was sensed as going low.

Bitwise XOR (Acc = Acc XOR Reg): This command performs a bitwise "XOR", exclusive or, on the Accumulator[10] value with the selected register value. The result is placed in Accumulator[10]. This means that the command compares each bit of both values and if either bit equals 1 or HIGH, the command places a 1 or HIGH in the result bit. If both bits equal 0 or 1, the result bit will be a 0 or LOW. The best example of this operation is the binary display.

Example: User[11] = 00000100 00000001 10001011 11001101, 0x0401 8BCD
Accumulator[10] = 00000000 00000010 11011100 10111010, 0x0002 DCBA
Bitwise XOR (Acc = Acc XOR Reg)
Accumulator[10] = Accumulator[10] OR User[11]

Result: Accumulator[10] = 00000100 00000011 01010111 01110111, 0x0403 5777

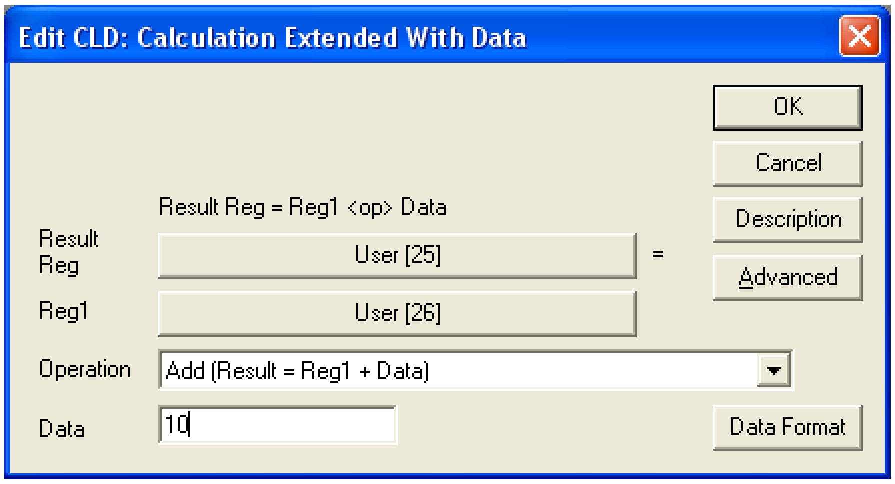
CLD:Calculation Extended with Data
CLX:Calculation Extended

Overview

The CLD/CLX commands provides basic math, logic and other functions using Data Registers with the second Parameter being a constant or a register. These command allows one source register, one constant (or a 2nd register) and one result register. The result register may be the same as the source register, if wanted. If the operation only needs a single register, then the source register is used.

The CLD and CLX commands are identical with the exception of Reg2 is substituted with Data. In the following examples, we will only use the CLD command.

For the examples, use these registers:
User[25] = Result
User[26] = Reg1


Edit CLD: Calculation Extended With Data

Result Reg = Reg1 <op> Data

Result Reg: User [25] =

Reg1: User [26]

Operation: Add (Result = Reg1 + Data)

Data: 10

CLD and CLX Operations

Absolute Value (Result = Abs(Reg1)): This command replaces the value from the selected register with a positive value of the same magnitude in the selected result register.

```
Example:   User[26] = -64
           Absolute Value (Result = Abs(Reg1))          User[25] = Abs(User[25])
Result:    User[25] = 64
```

Add (Result = Reg1 + Data): This command adds the value in selected register to the data and stores the result in the selected result register.

```
Example 1: User[26] = 10
           Data = 25
           Add (Result = Reg1 + Data)                   User[25] = User[26] + Data
Result:    User[25] = 35
```

Add (Result = HI(Reg1) + Data): This command is used to manipulate word boundary data present in some special registers. It adds the value from the data to the value in the higher word of a selected 32-bit selected register and stores the result in the selected result register. The best example of this operation is the binary display.

```
Example:    User[26] = 10
           Add (Result = HI(Reg1) + Data)            User[25] = HI(User[26]) + Data

           User[26] = 00000000 00001010 00000000 00000000, 0x000A 0000
                                        00000000 00001010  High word isolated, shifted
           Data     = 00000000 00000000 00000000 01001100, 0x4C
Result:    User[25] = 00000000 00000000 00000000 01010110, 0x0000 0056
```

Add (Result = LO(Reg1) + Data): This command adds the value from the data to the value in the lower word of a selected 32-bit selected register and stores the result in the selected result register. The best example of this operation is the binary display.

```
Example:    User[26] = 10
           Add (Result = LO(Reg1) + Data)            User[25] = HI(User[26]) + Data

           User[26] = 00000010 00000000 00000000 00001010, 0x0200 000A
           Data     = 00000000 00000000 00000000 01001100, 0x0000 004C
Result:    User[25] = 00000000 00000000 00000000 01010110, 0x0000 0056
```

Bitwise AND (Result = Reg1 AND Data): This command performs a bitwise "AND" on the data value with the selected register value. The result stores in the selected result register. This means the command compares each bit of both values and if both bits equal 1 or HIGH, the command places a 1 or HIGH in the result bit. Any other combination places a 0 or LOW. The best example of this operation is the binary display.

```
Example:    User[26] = 00000001 00000000 10101011 11001101, 0x0100 ABCD
            Data     = 00000001 00000000 11011100 10111010, 0x0100 DCBA
            Bitwise AND (Result = Reg1 AND Data)        User[25] = User[26] AND Data
Result:     User[25] = 00000001 00000000 10001000 10001000, 0x0100 8888
```

Copy (Result = Reg1): This command copies the value of a selected register to the selected result register.

```
Example:    Copy(Reg1)      User[25] = User[26]
Result:     User[25] now contains the value stored in User[26]
```

Copy Word, Sign Extend (Result = HI(Reg1)): This command will copy the high word (bits 16-32) of the selected 32-bit selected register to the low word (bits 0-15) of the selected register. The best example of this operation is the binary display.

```
Example:    User[26] = 10101011 11001101 00000000 00000011, 0xABCD 0003
            User[25] = 00000000 00000000 00000000 00000000, 0x0
            Copy Word, Sign Extend (Result = HI(Reg1))      LO(User[25]) = HI(User[26])
Result:     User[25] = 00000000 00000000 10101011 11001101, 0x0000 ABCD
```

Copy Word, Sign Extend (Result = LO(Reg1)): This command will copy the low word (bits 0-15) of the selected 32-bit selected register to the low word (bits 0-15) of the selected register. The best example of this operation is the binary display.

```
Example:    User[26] = 00000011 00000000 10101011 11001101, 0x0003 ABCD
            User[25] = 00000000 00000000 00000000 00000000, 0x0
            Copy Word, Sign Extend (Result = LO(Reg1))      LO(User[25]) = LO(User[26])
Result:     User[25] = 00000000 00000000 10101011 11001101, 0xABCD
```

Copy Word (Result = LO(Reg1) << 16 + LO(Data)): This command will shift the low word of the selected register to the left by 16 bits, making it the high word. Then the high word adds to the low word of the data. The best example of this operation is the binary display. This command is used to stack two 16 bit Words into a 32 bit word.

```
Example:    User[26] = 00000001 00000000 11001000 00000000, 0x0100 C800
                       11001000 00000000 00000000 00000000, [26] Data shifted left 16
            Data     = 00000000 00000000 00000000 11010011, 0x0000 00D3

            Copy Word (Result = LO(Reg1) << 16 + LO(Data))
                                                  User[25] = LO(User[26])<<16+ LO(Data)
Result:     User[25] = 11001000 00000000 00000000 11010011, 0xC800 00D3
```

Copy Word (Result = HI(Reg1) << 16 + LO(Data)): This command will shift the high word of the selected register to the left by 16 bits, making it the high word. Then the high word adds to the low word of the data. The best example of this operation is the binary display. This command is used to stack two 16 bit registers into a 32 bit result.

```
Example:    User[26] = 11001000 00000000 00000000 00001000, 0xC800 0008
            Data     = 00000000 00000000 00000000 11010011, 0x0000 00D3
            Copy Word (Result = HI(Reg1) << 16 + LO(Data))
                                                 User[25] = HI(User[26])<<16+ LO(Data)
Result:     User[25] = 11001000 00000000 00000000 11010011, 0xC800 00D3
```

Copy Word (Result = HI(Reg1) << 16 + HI(Data)): This command will shift the high word of the selected register to the left by 16 bits, making it the high word. Then the high word adds to the high word of the data. The best example of this operation is the binary display. This command is used to combine two 16 bit words into a single 32 bit word

```
Example:    User[26] = 11001000 00000000 00000000 00000000, 0xC800 0000
            User[26] = 11001000 00000000 00000000 00000000, 0xC800 xxxx
            Data     = 00000000 11010011 00000000 00000000, 0x00D3 0000
            Shifted  = 00000000 00000000 00000000 11010011
            Copy Word (Result = HI((Reg1) << 16 + HI(Data))
                                                 User[25] = HI(User[26])<<16+ HI(Data))
Result:     User[25] = 11001000 00000000 00000000 11010011, 0xC800 00D3
```

Div – Signed 32/16 bit (Result = Reg1 / LO(Data)): This command divides the signed selected register value by the signed 16-bit data value. The selected register value can be a signed 32-bit integer, but if the value is greater than a 16-bit integer, it ignores the upper word (bits 16-31). Note that this is not a floating-point calculation. The result is always an integer and any remainder is lost.

Value Ranges:
Register 1 = -2,147,483,648 to 2,147,483,647 (or $-2^{(31)}$ to $2^{(31)}$ - 1)
Data = 0 - 65535 (or 0 to 2^{16} - 1)

```
Example:    User[26] = 100
            Data = 10
            Div – Signed 32/16 bit (Result = Reg1 / LO(Data))     User[25] = User[26] / Data
Result:     User[25] = 10
```

Div – Signed 32/16 bit (Result = Data / LO(Reg1)): This command divides the signed data value by the signed 16-bit selected register value. The data value can be a signed 32-bit integer, but if the value is greater than a 16-bit integer, it ignores the upper word (bits 16-31). Note that this is not a floating-point calculation. The result is always an integer and any remainder is lost.

Value Ranges:
Register 1 = -2,147,483,648 to 2,147,483,647 (or $-2^{(31)}$ to $2^{(31)} - 1$)
Data = 0 - 65535 (or 0 to $2^{(16)} - 1$)

Example:	User[26] = 100	
	Data = 100	
	Div – Signed 32/16 bit (Result = Data / LO(Reg1))	*User[25] = Data / User[26]*
Result:	User[25] = 10	

Div – Unsigned 64/32 bit (Result = Reg1 / Data): This command divides the unsigned selected register value by the unsigned 32-bit data value. The dividend is composed of two user registers. The named register is the upper long word (top 32 bits), the following register (one higher) is used for the lower 32 bits of the 64 bit unsigned integer. The divisor is a 32 bit unsigned number.
Value Ranges:
Register 1 = 0 to 18,446,744,073,709,551,615 (or 0 to $2^{(64)} - 1$)
Data = 0 to 4,294,967,295 (or 0 to $2^{(32)} - 1$)

Example :	User[26]:[27] = 100,000,000,000,000	
	Data = 100,000,000	
	Div – Unsigned 64/32 bit (Result = Reg1:2 / Data)	
		User[25] = User[26]:[27] / Data
Result:	User[25] = 1,000,000	

Max (Result = Max of Reg1 or Data): This command will compare the selected register with the data. The larger (or least negative) of the two will be used as the result.

Example:	User[26] = 50	
	Data = 10	
	Max (Result = Max of Reg1 or Data)	*User[26] > Data*
Result:	User[25] = 50	

Min (Result = Min of Reg1 or Data): This command will compare the selected register with the data. The smaller (or most negative) of the two will be used as the result.

Example:	User[26] = 50	
	Data = 10	
	Min (Result = Min of Reg1 or Data)	*User[26] > Data*
Result:	User[25] = 10	

Modulo – 32 % 16 bit (Result = Data % LO(Reg1)): This command will modulo the signed data value by the signed 16-bit selected register value. The data value can be a signed 32-bit integer, but the modulo is limited to a 16-bit integer, (data in the upper word (bits 16-31) is ignored). The returned result is the remainder from the division of the two registers.

Value Ranges:
Register 1 = -2,147,483,648 to 2,147,483,647 (or $-2^{(31)}$ to $2^{(31)} - 1$)
Data = 0 - 65535 (or 0 to $2^{16} - 1$)

Example:	User[26] = 15
	Data = 100
	Modulo – 32 % 16 bit (Result = Data % LO(Reg1)) *User[25] = Data % User[26]*
Result:	User[25] = 10

Modulo – 32 % 16 bit (Result = LO(Reg1) % Data): This command will modulo the signed selected register value by the signed 16-bit data value. The selected register value can be a signed 32-bit integer, but modulo operator is limited to a 16-bit integer, (the upper word (bits 16-31) are ignored). The returned result is the remainder from the division of the two registers.

Value Ranges:
Register 1 = -2,147,483,648 to 2,147,483,647 (or $-2^{(31)}$ to $2^{(31)} - 1$)
Data = 0 - 65535 (or 0 to $2^{16} - 1$)

Example:	User[26] = 100
	Data = 15
	Modulo – 32 % 16 bit (Result = LO(Reg1) % Data) *User[25] = User[26] % Data*
Result:	User[25] = 10

Mult – Unsigned (Result = Reg1 * Data): This command multiplies the unsigned selected register value with the unsigned data value. The selected result register stores the unsigned result.

Value Ranges:
Register 1 = 0 to 4294967295 (or 0 to $2^{(32)} - 1$)
Data = 0 to 65535 (or 0 to $2^{(16)} - 1$)

Example:	User[26] = 5	
	Data = 10	
	Mult – Unsigned (Result = Reg1 * Data)	*User[25] = User[26] * Data*
Result:	User[25] = 50	

Mult – Signed (Result = Reg1 * Data): This command multiplies the signed selected register value with the signed data value. The selected result register stores the signed result.

Value Ranges:
Register 1 = -2,147,483,648 to 2,147,483,647 (or $-2^{(31)}$ to $2^{(31)} - 1$)
Data = -66536 to 65535 (or -2^{16} to $2^{16} - 1$)

Example: User[26] = -5
Data = 10
Mult – Signed (Result = Reg1 * Data) *User[25] = User[26] * Data*
Result: User[25] = -50

Mult – Signed 64 Bit (Result = (Reg1 * Data) >> 16): This command will multiply the signed 32-bit selected register with the signed 32-bit data. The data shifts right by 16 bits (or is divided by 65536). This was implemented for the goal of multiplying mixed fraction numbers (ex 12.1234) to prevent the loss of the decimal places in the calculation, but the result register returns as an integer. The middle 32 bits of the 64 bit product is returned.
Practical use: Multiply the contents of Register 1 by 254.3467
Step 1, multiply 254.3467 by 65536 and round to the nearest integer: 16668865
(the truncation makes this approximately 254.34669494; resolution is limited in fixed point)
Step 2 use this as the multiplicand.

Value Ranges:
Register 1 = -2,147,483,648 to 2,147,483,647 (or $-2^{(31)}$ to $2^{(31)} - 1$)
Data = -66536 to 65535 (or -2^{16} to $2^{16} - 1$)

Example: User[26] = 1000
Data = 16668865
Mult – Signed 64 Bit (Result = Reg1 * Data) >> 16)
*User[25] = (User[26] * Data) >> 16*
Result: User[25] = 254346

Mult – Unsigned 64 Bit (Result(U64) = Reg1 * Data): This command will multiply the unsigned 32-bit selected register with the unsigned 32-bit data. The result is stored in two successive 32 bit user registers to form a 64 bit unsigned number. This allows very large calculations to be preformed without overflow. The result register selected in the command is the first (upper) long word of the result. The lower long word is stored in the following (next highest) register.

Value Ranges:
Register 1 = 0 to 4294967295 (or 0 to $2^{(32)}$ - 1)
Data = 0 to 4294967295 (or 0 to $2^{(32)}$ - 1)

Example:	User[26] = 1,000,000
	Data = 1,000,000,000
	Mult – Unsigned 64 Bit (Result(U64) = Reg1 * Data)
	*User[28]:[29] = (User[26] * Data)*
Result:	*User[28]:[29*= 1,000,000,000,000,000

Negative (Result = -Reg1): This command will multiply the selected register by –1 and store the value in the selected result register.

Example:	User[26] = 100	
	Negative (Result = -Reg1)	*User[25] = -User[26]*
Result:	User[25] = -100	

Bitwise OR (Result = Reg1 OR Data): This command performs a bitwise "OR" on the selected register value with the data value. The result is placed in the selected result register. This means that the command compares each bit of both values and if either or both bits equal 1 or HIGH, the command places a 1 or HIGH in the result bit. Only if both bits equal 0, the result bit will be a 0 or LOW. The best example of this operation is the binary display.

Example:	User[26]	= 00000011 00000000 10001011 11001101, 0x0300 8BCD	
	Data	= 00000000 00000001 11011100 10111010, 0x0001 DCBA	
	Bitwise OR (Result = Reg1 OR Data)		*User[25] = User[26] OR Data*
Result:	User[25]	= 00000011 00000001 11011111 11111111, 0x0301 DFFF	

Queue Commands

The user registers may be configured into one or more queues by use of the Queue subcommands with in the CLX and CLD commands. Each queue is implemented as a circular buffer. Both ends of the queue are available for reading and writing allowing both FIFO (first in first out) and LIFO (Last in first out) functions to be implemented. Two overhead registers are associated with each Queue to hold the queue size, current size, and head and tail pointers. The queue is initialized via a Queue Init Command (below). Each of the Queue access commands comes in two varieties. The first set will cause a program error and stop the program if the queue is empty on a read or full on a write. The second set uses the arithmetic flag bits in ISW (Internal Status Word) to indicate the success (Sets the Zero flag – ISW Bit 1) or the failure (sets the Neg flag – ISW bit 3) of the operation. The user program should then test these bits using the JMP command after each operation.

The Head points to where the next Head Push will occur. The Tail points to where the next Tail Pop will occur. Thus, the Push Head operation stores data at the location pointed to by the Head pointer, then increments and modulos the Head pointer. The Pop Head operation decrements and modulos the Head pointer, and then returns the data using the modified head pointer. The Pop Tail returns the data as pointed to by the Tail pointer, then increments and modulos the Tail pointer. The Push Tail decrements and modulos the tail pointer and then stores the data at the new pointer location. The modulo operation in each case keeps the pointer within the queue. The queue size is incremented with each push and decremented with each pop, with error checking to see that the queue does not overflow or underflow.
A Push Head/Pop Head forms a LIFO (Last in First Out) register, while a Push Head/Pop Tail forms a FIFO (First in First Out) register. The user is free to read or write from either end of the queue, as well as to non-destructively read, using offsets from either end of the queue.

Queue Init (Reg1:Base, Data:Size): This command initializes a queue starting at the base register 1, the data value congfigures the size of the queue. The total storage is 2 + the requested queue size. The base register holds the allocated queue size in the upper word, and the current queue size in the lower word. The base register+1 holds the Head pointer in the upper word and the Tail pointer in the lower word. The pointers are relative to Base+2. These registers should not be modified by the user to prevent disruption of the queue operation. Base+2 though Base+2+(Size-1) hold the queue data.
Example programs for these commands are in the directory:
QuickControl\QCI Examples\Data Registers.

Example:

Result register:	User[10]	*(Result register equal the queue size) (not used by the queue)*
Reg1:	User[27]	*(Register 27 starting register of queue)*
Data:	Data = 2	*(Queue size)*

User[10]=2
User[27]=Max size: Current size
User[28]=Head pointer: Tail pointer
User[29]=data 2
User[30]=data 1

Queue-Cmd Err Push Head (Reg1:Head<=Data): Register 1 is the base register of the queue and the data value contains the data being pushed on the head of the queue. The data pushed on the queue. A Command Error is generated if the queue overflows. The destination register returns the remaining queue size.

Example:

Result register:	User[10]	*(Result register equal the remaining queue size)*
Reg1:	User[27]	(*Register 27 starting register of queue – assume queue size is 3)*
Data:	10	(*Data value pushed on stack)*

User[10]=xx
User[27]=reserved
User[28]=reserved
User[29]=data 1(Head, Tail)
User[30]=data 2
User[31]=data 3

Push Head
Data = 10

$\longrightarrow$

User[10]=2
User[27]=reserved
User[28]=reserved
User[29]=10 (Tail)
User[30]=data 2 (Head)
User[31]=data 3

Push Head
Data = 20

User[10]=1
User[27]=reserved
User[28]=reserved
User[29]=10 (Tail)
User[30]=20
User[31]=data 3 (Head)

Push Head
Data = 30

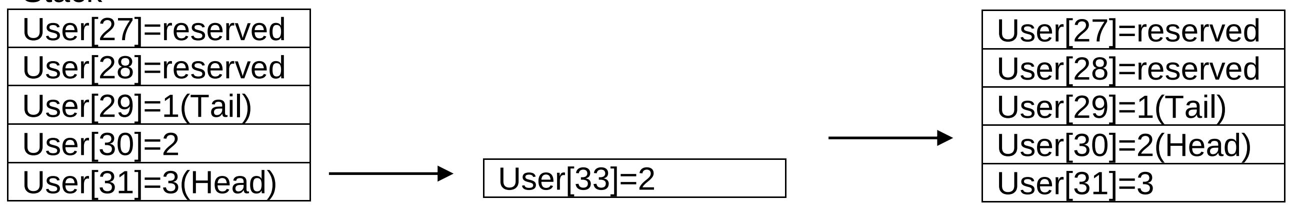
User[10]=0
User[27]=reserved
User[28]=reserved
User[29]=10 (Tail)(Head)
User[30]=20
User[31]=30

Queue-Cmd Err Pop Head (Reg1:Head=>Result): Register 1 is the base register of the queue. The head of the queue will decrement to the next register. The data from that register (new head) of the queue is stored into the result register. A command error is generated if the queue was empty.

Example:

Result register:	User[33]	*(Result register equal the data from the head queue)*
Reg1:	User[27]	*(Register 27 starting register of queue, two values are in the queue)*

Stack



User[27]=reserved
User[28]=reserved
User[29]=1(Tail)
User[30]=2
User[31]=3(Head)

User[33]=2

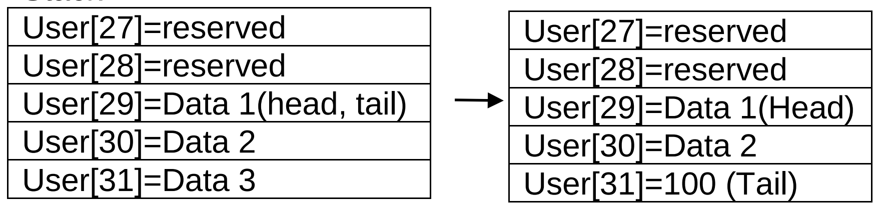
User[27]=reserved
User[28]=reserved
User[29]=1(Tail)
User[30]=2(Head)
User[31]=3

Queue-Cmd Err Push Tail (Reg1:Tail<=Data): Register 1 is the base register of the queue and the data value contains the data being pushed on the tail of the queue. A command error acknowledges when the queue is overflowed or empty. The destination register stores the remaining queue size.

Example:

Result register:	User[10]	*(Result register equal the remaining queue size)*
Reg1:	User[27]	*(Register 27 starting register of queue – assume queue size =3)*
Data:	100	(*Data value pushed on stack)*

Stack



User[27]=reserved
User[28]=reserved
User[29]=Data 1(head, tail)
User[30]=Data 2
User[31]=Data 3

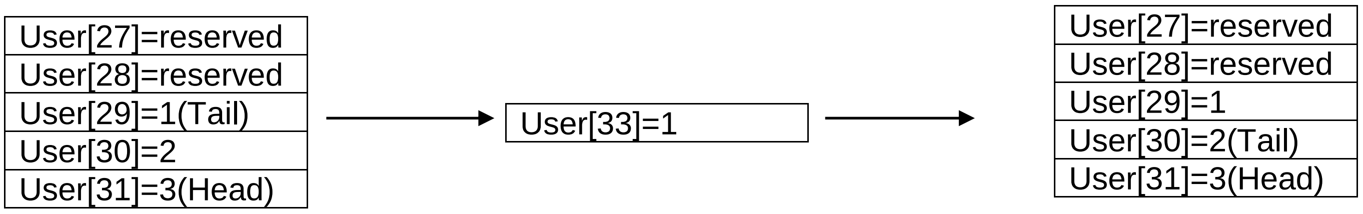
User[27]=reserved
User[28]=reserved
User[29]=Data 1(Head)
User[30]=Data 2
User[31]=100 (Tail)

Queue-Cmd Err Pop Tail (Reg1:Tail=>Result): Register 1 is the base register of the queue. The data from the tail of the queue is stored into the result register. The tail of the queue will move to the next register above. A command error acknowledges when the queue is empty.

Example:

Result register:	User[33]	*(Result register equal the data from the head queue)*
Reg1:	User[27]	*(Rregister 27 starting register of queue)*

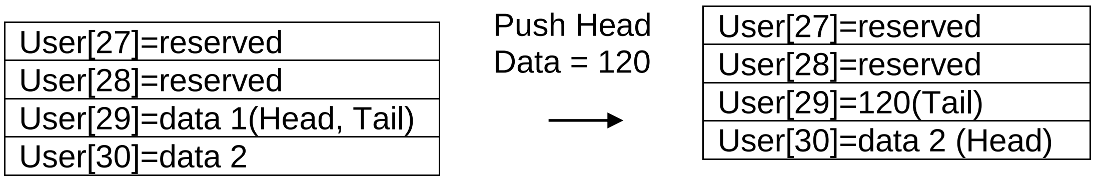
Stack



Queue-Push Head (Reg1:Head<=Data): Register 1 is the base register of the queue and the data value contains the data being pushed on the head of the queue. The data pushed on the queue will be stored at the head pointer location. If the queue is full, a command error will not be generated, but ISW bit 3 (Negative) is set. If the queue operation is successful, then ISW bit 1 (Zero) is set. The destination register stores the remaining queue size.

Example:

Result register:	User[10]	*(Result register equal the remaining queue size)*
Reg1:	User[27]	*(Register 27 starting register of queue)*
Data:	120	(*Data value pushed on stack)*

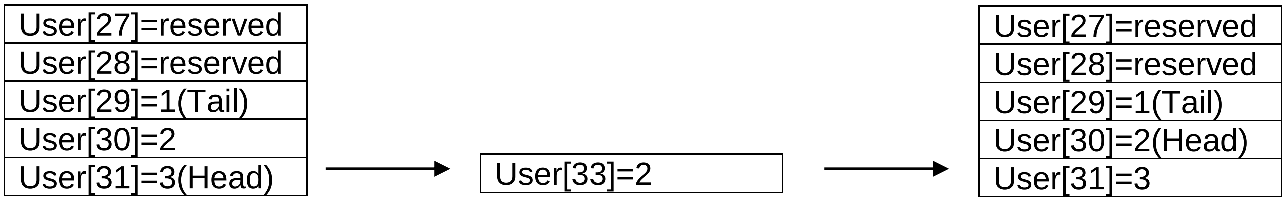


Queue- Pop Head (Reg1:Head=>Result): Register 1 is the base register of the queue. The head pointer will be decremented to point to the data, the data from the head of the queue is stored into the result register. If the queue was empty a command error will not be generated, but ISW bit 3 (Negative) is set to indicate the error. If the queue reports no error, then ISW bit 1 (Zero) is set to indicate success.

Example:

Result register:	User[33]	*(Result register equal the data from the head queue)*
Reg1:	User[27]	*(Register 27 starting register of queue)*

Stack

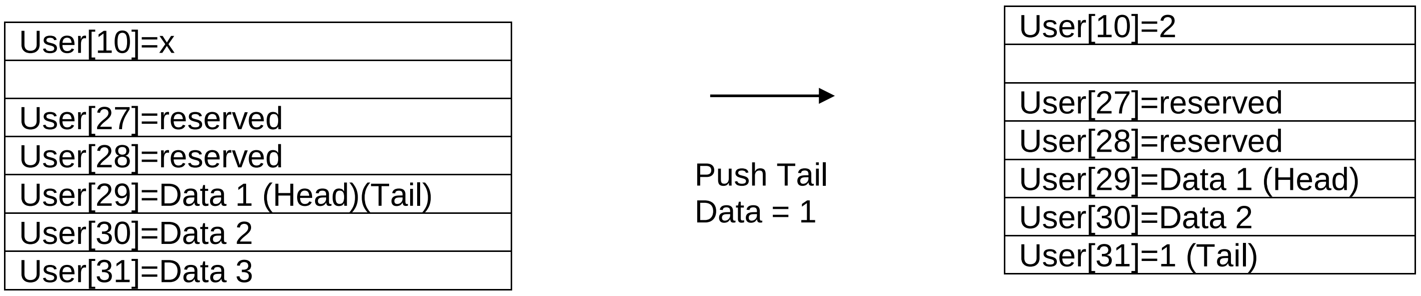


Queue- Push Tail (Reg1:Tail<=Data): Register 1 is the base register of the queue and the data value contains the data being pushed on the tail of the queue. The tail pointer is decremented (and moduloed). Data is written to the new tail pointer location. If the queue was full a command error will not be generated, but ISW bit 3 (Negative) is set. If the queue reports no error, then ISW bit 1 (Zero) is set. The destination register stores the remaining queue size.

Example:

Result register:	User[10]	*(Result register equal the remaining queue size)*
Reg1:	User[27]	*(Register 27 starting register of queue, assume a queue size of 3)*
Data:	Data	(*Data value pushed on stack)*

Stack

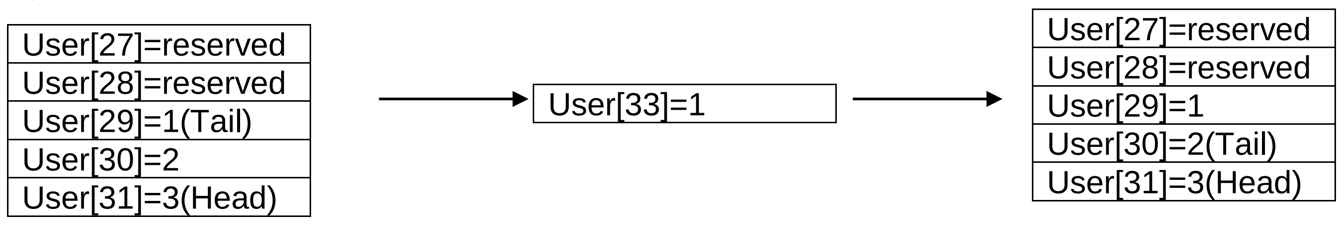


Queue- Pop Tail (Reg1:Tail=>Result): Register 1 is the base register of the queue. The data from the tail of the queue is stored into the result register. The tail of the queue will move to the next register above. If the queue was empty a command error will not be generated, but ISW bit 3 (Negative) is set to indicate the error. If the queue reports no error, then ISW bit 1 (Zero) is set to indicate success.

Example:

Result register:	User[33]	*(Result register equal the data from the head queue)*
Reg1:	User[27]	*(Register 27 starting register of queue)*

Stack

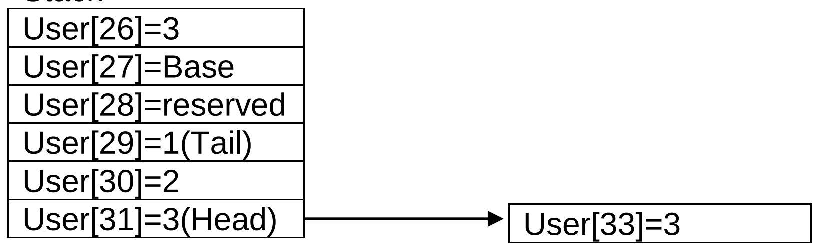


Queue- Read Element (Reg1:Result=Data): Register 1 is the base register of the queue. The data element copies to the result register. The element read is determined by the value stored in the data value. If the value is Positive, the element read is counted from the head of the queue. If the value is Negative, the element read is counted from the tail of the queue. Zero reads non-destructively from the head of the queue, Negative 1 reads non-destructively from the tail of the queue, etc.

Example:

Result register:	User[33]	*(Result register stores the read element data)*
Reg1:	User[27]	*(Register 27 starting register of queue)*
Data:	0	*(Data determines the element to be read)*

Stack



Sub (Result = Reg1 – Data): This command subtracts the value in the data value from the selected register and stores the result in the selected result register.

Example 1:	User[26] = 25	
	Data = 10	
	Sub (Result = Reg1 – Data)	*User[25] = User[26] – Data*
Result:	User[25] = 15	

Sub (Result = Data – Reg1): This command subtracts the selected register from the data value and stores the result in the selected result register.

Example 1:	User[26] = 25	
	Data = 10	
	Sub (Result = Data – Reg1)	*User[25] = Data – User[26]*
Result:	User[25] = -15	

Bitwise XOR (Result = Reg1 XOR Data): This command performs a bitwise “XOR”, exclusive or, on the selected register value with the data value. The result stores in the selected result register. This means the command compares each bit of both values and if either bit equals 1 or HIGH, the command places a 1 or HIGH in the result bit. If both bits equal 0 or 1, the result bit will be a 0 or LOW. The best example of this operation is the binary display.

Example:	User[26]	= 00000011 00000000 10001011 11001101, 0x0300 8BCD
	Data	= 00000001 00000000 11011100 10111010, 0x0100 DCBA
	Bitwise XOR (Result = Reg1 XOR Data)	*User[25] = User[26] XOR Data*
Result:	User[25]	= 00000010 00000000 01010111 01110111, 0x0200 5777