Lecture Notes on Assembly Language - J. Vaughan

18. Shifts and Rotates

`SHL`, `SHR`: Bitwise Logical Shifts

```
SHL r/m8,1                    ; D0 /4                  [8086]
SHL r/m8,CL                   ; D2 /4                  [8086]
SHL r/m8,imm8                 ; C0 /4 ib               [186]
SHL r/m16,1                   ; o16 D1 /4              [8086]
SHL r/m16,CL                  ; o16 D3 /4              [8086]
SHL r/m16,imm8                ; o16 C1 /4 ib           [186]
SHL r/m32,1                   ; o32 D1 /4              [386]
SHL r/m32,CL                  ; o32 D3 /4              [386]
SHL r/m32,imm8                ; o32 C1 /4 ib           [386]
SHR r/m8,1                    ; D0 /5                  [8086]
SHR r/m8,CL                   ; D2 /5                  [8086]
SHR r/m8,imm8                 ; C0 /5 ib               [186]
SHR r/m16,1                   ; o16 D1 /5              [8086]
SHR r/m16,CL                  ; o16 D3 /5              [8086]
SHR r/m16,imm8                ; o16 C1 /5 ib           [186]
SHR r/m32,1                   ; o32 D1 /5              [386]
SHR r/m32,CL                  ; o32 D3 /5              [386]
SHR r/m32,imm8                ; o32 C1 /5 ib           [386]
```

`SHL` and `SHR` perform a logical shift operation on the given source/destination (first) operand. The vacated bits are filled with zero.
A synonym for `SHL` is `SAL` (see section B.4.283). NASM will assemble either one to the same code, but NDISASM will always disassemble that code as `SHL`.
The number of bits to shift by is given by the second operand. Only the bottom five bits of the shift count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a `C1` byte) form of `SHL foo,1` by using a `BYTE` prefix: `SHL foo,BYTE 1` . Similarly with `SHR`.

`SAL`, `SAR`: Bitwise Arithmetic Shifts

```
SAL r/m8,1                    ; D0 /4                  [8086]
SAL r/m8,CL                   ; D2 /4                  [8086]
SAL r/m8,imm8                 ; C0 /4 ib               [186]
SAL r/m16,1                   ; o16 D1 /4              [8086]
SAL r/m16,CL                  ; o16 D3 /4              [8086]
SAL r/m16,imm8                ; o16 C1 /4 ib           [186]
SAL r/m32,1                   ; o32 D1 /4              [386]
SAL r/m32,CL                  ; o32 D3 /4              [386]
SAL r/m32,imm8                ; o32 C1 /4 ib           [386]
SAR r/m8,1                    ; D0 /7                  [8086]
SAR r/m8,CL                   ; D2 /7                  [8086]
SAR r/m8,imm8                 ; C0 /7 ib               [186]
SAR r/m16,1                   ; o16 D1 /7              [8086]
SAR r/m16,CL                  ; o16 D3 /7              [8086]
SAR r/m16,imm8                ; o16 C1 /7 ib           [186]
SAR r/m32,1                   ; o32 D1 /7              [386]
SAR r/m32,CL                  ; o32 D3 /7              [386]
SAR r/m32,imm8                ; o32 C1 /7 ib           [386]
```

`SAL` and `SAR` perform an arithmetic shift operation on the given source/destination (first) operand.
The vacated bits are filled with zero for `SAL`, and with copies of the original high bit of the source operand for `SAR`.

SAL is a synonym for SHL (see section B.4.290). NASM will assemble either one to the same code, but NDISASM will always disassemble that code as SHL.
The number of bits to shift by is given by the second operand. Only the bottom five bits of the shift count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a C1 byte) form of SAL foo,1 by using a BYTE prefix: SAL foo,BYTE 1 . Similarly with SAR.

ROL, ROR: Bitwise Rotate

```
ROL r/m8,1                    ; D0 /0                 [8086]
ROL r/m8,CL                   ; D2 /0                 [8086]
ROL r/m8,imm8                 ; C0 /0 ib              [186]
ROL r/m16,1                   ; o16 D1 /0             [8086]
ROL r/m16,CL                  ; o16 D3 /0             [8086]
ROL r/m16,imm8                ; o16 C1 /0 ib          [186]
ROL r/m32,1                   ; o32 D1 /0             [386]
ROL r/m32,CL                  ; o32 D3 /0             [386]
ROL r/m32,imm8                ; o32 C1 /0 ib          [386]
ROR r/m8,1                    ; D0 /1                 [8086]
ROR r/m8,CL                   ; D2 /1                 [8086]
ROR r/m8,imm8                 ; C0 /1 ib              [186]
ROR r/m16,1                   ; o16 D1 /1             [8086]
ROR r/m16,CL                  ; o16 D3 /1             [8086]
ROR r/m16,imm8                ; o16 C1 /1 ib          [186]
ROR r/m32,1                   ; o32 D1 /1             [386]
ROR r/m32,CL                  ; o32 D3 /1             [386]
ROR r/m32,imm8                ; o32 C1 /1 ib          [386]
```

ROL and ROR perform a bitwise rotation operation on the given source/destination (first) operand.
Thus, for example, in the operation ROL AL,1 , an 8–bit rotation is performed in which AL is shifted left by 1 and the original top bit of AL moves round into the low bit.
The number of bits to rotate by is given by the second operand. Only the bottom five bits of the rotation count are considered by processors above the 8086.
You can force the longer (286 and upwards, beginning with a C1 byte) form of ROL foo,1 by using a BYTE prefix: ROL foo,BYTE 1 . Similarly with ROR.

RCL, RCR: Bitwise Rotate through Carry Bit

```
RCL r/m8,1                    ; D0 /2                 [8086]
RCL r/m8,CL                   ; D2 /2                 [8086]
RCL r/m8,imm8                 ; C0 /2 ib              [186]
RCL r/m16,1                   ; o16 D1 /2             [8086]
RCL r/m16,CL                  ; o16 D3 /2             [8086]
RCL r/m16,imm8                ; o16 C1 /2 ib          [186]
RCL r/m32,1                   ; o32 D1 /2             [386]
RCL r/m32,CL                  ; o32 D3 /2             [386]
RCL r/m32,imm8                ; o32 C1 /2 ib          [386]
RCR r/m8,1                    ; D0 /3                 [8086]
RCR r/m8,CL                   ; D2 /3                 [8086]
RCR r/m8,imm8                 ; C0 /3 ib              [186]
RCR r/m16,1                   ; o16 D1 /3             [8086]
RCR r/m16,CL                  ; o16 D3 /3             [8086]
RCR r/m16,imm8                ; o16 C1 /3 ib          [186]
RCR r/m32,1                   ; o32 D1 /3             [386]
RCR r/m32,CL                  ; o32 D3 /3             [386]
RCR r/m32,imm8                ; o32 C1 /3 ib          [386]
```

RCL and RCR perform a 9–bit, 17–bit or 33–bit bitwise rotation operation, involving the given source/destination (first) operand and the carry bit. Thus, for example, in the operation RCL AL,1 , a 9–bit rotation is performed in which AL is shifted left by 1, the top bit of AL moves into the carry flag, and the original value of the carry flag is

placed in the low bit of `AL`.
The number of bits to rotate by is given by the second operand. Only the bottom five bits of the rotation count are considered by processors above the 8086.
You can force the longer (286 and upwards, beginning with a `C1` byte) form of `RCL foo,1` by using a `BYTE` prefix: `RCL foo,BYTE 1`. Similarly with `RCR`.

Shifting the DX:AX register pair

Routine to shift the DX:AX registers **LEFT** by a number of bits specified in the ECX register. The bits to be shifted are already in the registers.
; Example: Shift **left** by 4 bits

```
        mov ecx, 4
shftlp  shl dx, 1       ; shift dx
        shl ax, 1       ; shift ax
        adc dx, 0       ; move the carry from ax into dx
        loop shftlp
```

Shift the DX:AX registers **RIGHT** by a number of bits specified in the ECX register.
; Example: Shift **right** by 4 bits

```
        mov ecx, 4
shftlp  shr ax, 1       ; shift ax
        shr dx, 1       ; shift dx
        jnc cntnu       ; no bit shifted from dx
        or ah, 0x40     ; move the carry from dx into ax
cntnu   loop shftlp
```

Shift the EDX:EAX registers **LEFT** by an amount specified in ECX.
; Example: Shift **left** by 4 bits

```
        mov ecx, 4
shftlp  shl edx, 1      ; shift dx
        shl eax, 1      ; shift ax
        adc edx, 0      ; move the carry from ax into dx
        loop shftlp
```

Shift the EDX:EAX registers **RIGHT** by an amount specified in ECX.
; Example: Shift **right** by 4 bits

```
        mov ecx, 4
shftlp  shr eax, 1      ; shift eax
        shr edx, 1      ; shift edx
        jnc cntnu       ; no bit shifted from edx
        or ax, 0x4000   ; move the carry from edx into eax
cntnu    loop shftlp
```

19. String Instructions

`MOVSB`, `MOVSW`, `MOVSD` : Move String

```
MOVSB                            ; A4                  [8086]
MOVSW                            ; o16 A5              [8086]
MOVSD                            ; o32 A5              [386]
```

`MOVSB` copies the byte at `[DS:SI]` or `[DS:ESI]` to `[ES:DI]` or `[ES:EDI]`. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `SI` and `DI` (or `ESI` and `EDI`).

The registers used are `SI` and `DI` if the address size is 16 bits, and `ESI` and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.
The segment register used to load from `[SI]` or `[ESI]` can be overridden by using a segment register name as a prefix (for example, `es movsb`). The use of `ES` for the store to `[DI]` or `[EDI]` cannot be overridden.
`MOVSW` and `MOVSD` work in the same way, but they copy a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.
The `REP` prefix may be used to repeat the instruction `CX` (or `ECX` – again, the address size chooses which) times.

LODSB , LODSW , LODSD : Load from String

```
LODSB                              ; AC                 [8086]
LODSW                              ; o16 AD             [8086]
LODSD                              ; o32 AD             [386]
```

`LODSB` loads a byte from `[DS:SI]` or `[DS:ESI]` into `AL`. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `SI` or `ESI`.
The register used is `SI` if the address size is 16 bits, and `ESI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.
The segment register used to load from `[SI]` or `[ESI]` can be overridden by using a segment register name as a prefix (for example, `ES LODSB`).
`LODSW` and `LODSD` work in the same way, but they load a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

STOSB , STOSW , STOSD : Store Byte to String

```
STOSB                              ; AA                 [8086]
STOSW                              ; o16 AB             [8086]
STOSD                              ; o32 AB             [386]
```

`STOSB` stores the byte in `AL` at `[ES:DI]` or `[ES:EDI]` , and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `DI` (or `EDI`).
The register used is `DI` if the address size is 16 bits, and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.
Segment override prefixes have no effect for this instruction: the use of `ES` for the store to `[DI]` or `[EDI]` cannot be overridden.
`STOSW` and `STOSD` work in the same way, but they store the word in `AX` or the doubleword in `EAX` instead of the byte in `AL`, and increment or decrement the addressing registers by 2 or 4 instead of 1.
The `REP` prefix may be used to repeat the instruction `CX` (or `ECX` – again, the address size chooses which) times.

CMPSB , CMPSW , CMPSD : Compare Strings

```
CMPSB                              ; A6                 [8086]
CMPSW                              ; o16 A7             [8086]
CMPSD                              ; o32 A7             [386]
```

`CMPSB` compares the byte at `[DS:SI]` or `[DS:ESI]` with the byte at `[ES:DI]` or `[ES:EDI]` , and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `SI` and `DI` (or `ESI` and `EDI`).

The registers used are `SI` and `DI` if the address size is 16 bits, and `ESI` and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.
The segment register used to load from `[SI]` or `[ESI]` can be overridden by using a segment register name as a prefix (for example, `ES CMPSB`). The use of `ES` for the load from `[DI]` or `[EDI]` cannot be overridden.
`CMPSW` and `CMPSD` work in the same way, but they compare a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.
The `REPE` and `REPNE` prefixes (equivalently, `REPZ` and `REPNZ`) may be used to repeat the instruction up to `CX` (or `ECX` – again, the address size chooses which) times until the first unequal or equal byte is found.

SCASB , SCASW , SCASD : Scan String

```
SCASB                             ; AE                [8086]
SCASW                             ; o16 AF            [8086]
SCASD                             ; o32 AF            [386]
```

`SCASB` compares the byte in `AL` with the byte at `[ES:DI]` or `[ES:EDI]` , and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `DI` (or `EDI`).
The register used is `DI` if the address size is 16 bits, and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.
Segment override prefixes have no effect for this instruction: the use of `ES` for the load from `[DI]` or `[EDI]` cannot be overridden.
`SCASW` and `SCASD` work in the same way, but they compare a word to `AX` or a doubleword to `EAX` instead of a byte to `AL`, and increment or decrement the addressing registers by 2 or 4 instead of 1.
The `REPE` and `REPNE` prefixes (equivalently, `REPZ` and `REPNZ`) may be used to repeat the instruction up to `CX` (or `ECX` – again, the address size chooses which) times until the first unequal or equal byte is found.