Lecture Notes on Assembly Language - J. Vaughan

13. Instructions and data (ctd)

From the NASM Manual:

AND: Bitwise AND

```
AND r/m8,reg8                 ; 20 /r               [8086]
AND r/m16,reg16               ; o16 21 /r           [8086]
AND r/m32,reg32               ; o32 21 /r           [386]
AND reg8,r/m8                 ; 22 /r               [8086]
AND reg16,r/m16               ; o16 23 /r           [8086]
AND reg32,r/m32               ; o32 23 /r           [386]
AND r/m8,imm8                 ; 80 /4 ib            [8086]
AND r/m16,imm16               ; o16 81 /4 iw        [8086]
AND r/m32,imm32               ; o32 81 /4 id        [386]
AND r/m16,imm8                ; o16 83 /4 ib        [8086]
AND r/m32,imm8                ; o32 83 /4 ib        [386]
AND AL,imm8                   ; 24 ib               [8086]
AND AX,imm16                  ; o16 25 iw           [8086]
AND EAX,imm32                 ; o32 25 id           [386]
```

`AND` performs a bitwise AND operation between its two operands (i.e. each bit of the result is 1 if and only if the corresponding bits of the two inputs were both 1), and stores the result in the destination (first) operand. The destination operand can be a register or a memory location. The source operand can be a register, a memory location or an immediate value.
In the forms with an 8–bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign–extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

CALL: Call Subroutine

```
CALL imm                      ; E8 rw/rd            [8086]
CALL imm:imm16                ; o16 9A iw iw        [8086]
CALL imm:imm32                ; o32 9A id iw        [386]
CALL FAR mem16                ; o16 FF /3           [8086]
CALL FAR mem32                ; o32 FF /3           [386]
CALL r/m16                    ; o16 FF /2           [8086]
CALL r/m32                    ; o32 FF /2           [386]
```

`CALL` calls a subroutine, by means of pushing the current instruction pointer (`IP`) and optionally `CS` as well on the stack, and then jumping to a given address.
`CS` is pushed as well as `IP` if and only if the call is a far call, i.e. a destination segment address is specified in the instruction. The forms involving two colon–separated arguments are far calls; so are the `CALL FAR mem` forms.
The immediate near call takes one of two forms (`call imm16/imm32` , determined by the current segment size limit. For 16–bit operands, you would use `CALL 0x1234`, and for 32–bit operands you would use `CALL 0x12345678` . The value passed as an operand is a relative offset.
You can choose between the two immediate far call forms (`CALL imm:imm`) by the use of the `WORD` and `DWORD` keywords: `CALL WORD 0x1234:0x5678`) or `CALL DWORD 0x1234:0x56789abc` .
The `CALL FAR mem` forms execute a far call by loading the destination address out of memory.
The address loaded consists of 16 or 32 bits of offset (depending on the operand size), and 16 bits of segment. The operand size may be overridden using `CALL`

`WORD FAR mem` or `CALL DWORD FAR mem`.
The `CALL r/m` forms execute a near call (within the same segment), loading the destination address out of memory or out of a register. The keyword `NEAR` may be specified, for clarity, in these forms, but is not necessary. Again, operand size can be overridden using `CALL WORD mem` or `CALL DWORD mem`.
As a convenience, NASM does not require you to call a far procedure symbol by coding the cumbersome `CALL SEG routine:routine`, but instead allows the easier synonym `CALL FAR routine`.
The `CALL r/m` forms given above are near calls; NASM will accept the `NEAR` keyword (e.g. `CALL NEAR [address]`), even though it is not strictly necessary.

`CBW`, `CWD`, `CDQ`, `CWDE`: Sign Extensions

```
CBW                           ; o16 98                 [8086]
CWDE                          ; o32 98                 [386]
CWD                           ; o16 99                 [8086]
CDQ                           ; o32 99                 [386]
```

All these instructions sign–extend a short value into a longer one, by replicating the top bit of the original value to fill the extended one.
`CBW` extends `AL` into `AX` by repeating the top bit of `AL` in every bit of `AH`. `CWDE` extends `AX` into `EAX`. `CWD` extends `AX` into `DX:AX` by repeating the top bit of `AX` throughout `DX`, and `CDQ` extends `EAX` into `EDX:EAX`.

`CLC`, `CLD`, `CLI`, `CLTS`: Clear Flags

```
CLC                           ; F8                     [8086]
CLD                           ; FC                     [8086]
CLI                           ; FA                     [8086]
CLTS                          ; 0F 06                [286,PRIV]
```

These instructions clear various flags. `CLC` clears the carry flag; `CLD` clears the direction flag; `CLI` clears the interrupt flag (thus disabling interrupts); and `CLTS` clears the task–switched (`TS`) flag in `CR0`.
To set the carry, direction, or interrupt flags, use the `STC`, `STD` and `STI` instructions. To invert the carry flag, use `CMC`.

`CMC`: Complement Carry Flag

```
CMC                           ; F5                     [8086]
```

`CMC` changes the value of the carry flag: if it was 0, it sets it to 1, and vice versa.

`CMP`: Compare Integers

```
CMP r/m8,reg8                 ; 38 /r                  [8086]
CMP r/m16,reg16               ; o16 39 /r              [8086]
CMP r/m32,reg32               ; o32 39 /r              [386]
CMP reg8,r/m8                 ; 3A /r                  [8086]
CMP reg16,r/m16               ; o16 3B /r              [8086]
CMP reg32,r/m32               ; o32 3B /r              [386]
CMP r/m8,imm8                 ; 80 /7 ib               [8086]
CMP r/m16,imm16               ; o16 81 /7 iw           [8086]
CMP r/m32,imm32               ; o32 81 /7 id           [386]
CMP r/m16,imm8                ; o16 83 /7 ib           [8086]
CMP r/m32,imm8                ; o32 83 /7 ib           [386]
CMP AL,imm8                   ; 3C ib                  [8086]
CMP AX,imm16                  ; o16 3D iw              [8086]
CMP EAX,imm32                 ; o32 3D id              [386]
```

`CMP` performs a 'mental' subtraction of its second operand from its first operand, and affects the flags as if the subtraction had taken place, but does not store the result of the subtraction anywhere.
In the forms with an 8–bit immediate second operand and a longer first operand, the

second operand is considered to be signed, and is sign–extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.
The destination operand can be a register or a memory location. The source can be a register, memory location or an immediate value of the same size as the destination.

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.

DEC: Decrement Integer

```
DEC reg16                     ; o16 48+r        [8086]
DEC reg32                     ; o32 48+r        [386]
DEC r/m8                      ; FE /1           [8086]
DEC r/m16                     ; o16 FF /1       [8086]
DEC r/m32                     ; o32 FF /1       [386]
```

`DEC` subtracts 1 from its operand. It does *not* affect the carry flag: to affect the carry flag, use
`SUB something,1` (see section B.4.305). `DEC` affects all the other flags according to the result.
This instruction can be used with a `LOCK` prefix to allow atomic execution.
See also `INC`.

DIV: Unsigned Integer Divide

```
DIV r/m8                      ; F6 /6           [8086]
DIV r/m16                     ; o16 F7 /6       [8086]
DIV r/m32                     ; o32 F7 /6       [386]
```

`DIV` performs unsigned integer division. The explicit operand provided is the divisor; the dividend
and destination operands are implicit, in the following way:
For `DIV r/m8` , `AX` is divided by the given operand; the quotient is stored in `AL` and the remainder in `AH`.
For `DIV r/m16` , `DX:AX` is divided by the given operand; the quotient is stored in `AX` and the remainder in `DX`.
For `DIV r/m32` , `EDX:EAX` is divided by the given operand; the quotient is stored in `EAX` and the remainder in `EDX`.
Signed integer division is performed by the `IDIV` instruction.

HLT: Halt Processor

```
HLT                                ; F4                [8086,PRIV]
```

HLT puts the processor into a halted state, where it will perform no more operations until restarted by an interrupt or a reset.
On the 286 and later processors, this is a privileged instruction.

IDIV: Signed Integer Divide

```
IDIV r/m8                          ; F6 /7              [8086]
IDIV r/m16                         ; o16 F7 /7          [8086]
IDIV r/m32                         ; o32 F7 /7          [386]
```

IDIV performs signed integer division. The explicit operand provided is the divisor; the dividend and destination operands are implicit, in the following way:
For IDIV r/m8 , AX is divided by the given operand; the quotient is stored in AL and the remainder in AH.
For IDIV r/m16 , DX:AX is divided by the given operand; the quotient is stored in AX and the remainder in DX.
For IDIV r/m32 , EDX:EAX is divided by the given operand; the quotient is stored in EAX and the remainder in EDX.
Unsigned integer division is performed by the DIV instruction.

IMUL: Signed Integer Multiply

```
IMUL r/m8                          ; F6 /5              [8086]
IMUL r/m16                         ; o16 F7 /5          [8086]
IMUL r/m32                         ; o32 F7 /5          [386]
IMUL reg16,r/m16                   ; o16 0F AF /r       [386]
IMUL reg32,r/m32                   ; o32 0F AF /r       [386]
IMUL reg16,imm8                    ; o16 6B /r ib       [186]
IMUL reg16,imm16                   ; o16 69 /r iw       [186]
IMUL reg32,imm8                    ; o32 6B /r ib       [386]
IMUL reg32,imm32                   ; o32 69 /r id       [386]
IMUL reg16,r/m16,imm8              ; o16 6B /r ib       [186]
IMUL reg16,r/m16,imm16             ; o16 69 /r iw       [186]
IMUL reg32,r/m32,imm8              ; o32 6B /r ib       [386]
IMUL reg32,r/m32,imm32             ; o32 69 /r id       [386]
```

IMUL performs signed integer multiplication. For the single–operand form, the other operand and destination are implicit, in the following way:
For IMUL r/m8 , AL is multiplied by the given operand; the product is stored in AX.
For IMUL r/m16 , AX is multiplied by the given operand; the product is stored in DX:AX .
For IMUL r/m32 , EAX is multiplied by the given operand; the product is stored in EDX:EAX .
The two–operand form multiplies its two operands and stores the result in the destination (first) operand. The three–operand form multiplies its last two operands and stores the result in the first operand.
The two–operand form with an immediate second operand is in fact a shorthand for the three–operand form, as can be seen by examining the opcode descriptions: in the two–operand form, the code /r takes both its register and r/m parts from the same operand (the first one).
In the forms with an 8–bit immediate operand and another longer source operand, the immediate operand is considered to be signed, and is sign–extended to the length of the other source operand.
In these cases, the BYTE qualifier is necessary to force NASM to generate this form of the instruction.
Unsigned integer multiplication is performed by the MUL instruction.

`IN`: Input from I/O Port

```
IN AL,imm8                ; E4 ib           [8086]
IN AX,imm8                ; o16 E5 ib       [8086]
IN EAX,imm8               ; o32 E5 ib       [386]
IN AL,DX                  ; EC              [8086]
IN AX,DX                  ; o16 ED          [8086]
IN EAX,DX                 ; o32 ED          [386]
```

`IN` reads a byte, word or doubleword from the specified I/O port, and stores it in the given destination register. The port number may be specified as an immediate value if it is between 0 and 255, and otherwise must be stored in `DX`. See also `OUT`.

`INC`: Increment Integer

```
INC reg16                 ; o16 40+r        [8086]
INC reg32                 ; o32 40+r        [386]
INC r/m8                  ; FE /0           [8086]
INC r/m16                 ; o16 FF /0       [8086]
INC r/m32                 ; o32 FF /0       [386]
```

`INC` adds 1 to its operand. It does *not* affect the carry flag: to affect the carry flag, use `ADD something,1` . `INC` affects all the other flags according to the result.
This instruction can be used with a `LOCK` prefix to allow atomic execution.
See also `DEC`.

`INSB`, `INSW`, `INSD`: Input String from I/O Port

```
INSB                      ; 6C              [186]
INSW                      ; o16 6D          [186]
INSD                      ; o32 6D          [386]
```

`INSB` inputs a byte from the I/O port specified in `DX` and stores it at `[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) `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.
`INSW` and `INSD` work in the same way, but they input a word or a doubleword instead of a byte, and increment or decrement the addressing register 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.
See also `OUTSB` , `OUTSW` and `OUTSD`.

`INT`: Software Interrupt

```
INT imm8                  ; CD ib           [8086]
```

`INT` causes a software interrupt through a specified vector number from 0 to 255.
The code generated by the `INT` instruction is always two bytes long: although there are short forms
for some `INT` instructions, NASM does not generate them when it sees the `INT` mnemonic. In order to generate single–byte breakpoint instructions, use the `INT3` or `INT1` instructions instead.

`INT3`, `INT1`, `ICEBP` , `INT01` : Breakpoints

```
INT1                      ; F1              [P6]
ICEBP                     ; F1              [P6]
INT01                     ; F1              [P6]
INT3                      ; CC              [8086]
INT03                     ; CC              [8086]
```

`INT1` and `INT3` are short one–byte forms of the instructions `INT 1` and `INT 3`.

They perform a similar function to their longer counterparts, but take up less code space. They are used as breakpoints by debuggers.
`INT1` , and its alternative synonyms `INT01` and `ICEBP` , is an instruction used by in–circuit
emulators (ICEs). It is present, though not documented, on some processors down to the 286, but is only documented for the Pentium Pro. `INT3` is the instruction normally used as a breakpoint by debuggers.
`INT3` , and its synonym `INT03` , is not precisely equivalent to `INT 3` : the short form, since it is designed to be used as a breakpoint, bypasses the normal `IOPL` checks in virtual–8086 mode, and also does not go through interrupt redirection.

`INTO`: Interrupt if Overflow

```
INTO                          ; CE                    [8086]
```

`INTO` performs an `INT 4` software interrupt (see section B.4.122) if and only if the overflow flag is set.

`IRET`, `IRETW` , `IRETD` : Return from Interrupt

```
IRET                          ; CF                    [8086]
IRETW                         ; o16 CF                [8086]
IRETD                         ; o32 CF                [386]
```

`IRET` returns from an interrupt (hardware or software) by means of popping `IP` (or `EIP`), `CS` and the flags off the stack and then continuing execution from the new `CS:IP` .
`IRETW` pops `IP`, `CS` and the flags as 2 bytes each, taking 6 bytes off the stack in total. `IRETD` pops `EIP` as 4 bytes, pops a further 4 bytes of which the top two are discarded and the bottom two go into `CS`, and pops the flags as 4 bytes as well, taking 12 bytes off the stack.
`IRET` is a shorthand for either `IRETW` or `IRETD` , depending on the default `BITS` setting at the time.

`Jcc`: Conditional Branch

```
Jcc imm                       ; 70+cc rb              [8086]
Jcc NEAR imm                  ; 0F 80+cc rw/rd        [386]
```

The conditional jump instructions execute a near (same segment) jump if and only if their
conditions are satisfied. For example, `JNZ` jumps only if the zero flag is not set.
The ordinary form of the instructions has only a 128–byte range; the `NEAR` form is a 386 extension
to the instruction set, and can span the full size of a segment. NASM will not override your choice
of jump instruction: if you want `Jcc NEAR` , you have to use the `NEAR` keyword.
The `SHORT` keyword is allowed on the first form of the instruction, for clarity, but is not necessary.
For details of the condition codes, see section B.2.2.

`JCXZ`, `JECXZ` : Jump if CX/ECX Zero

```
JCXZ imm                      ; a16 E3 rb             [8086]
JECXZ imm                     ; a32 E3 rb             [386]
```

JCXZ performs a short jump (with maximum range 128 bytes) if and only if the contents of the CX register is 0. JECXZ does the same thing, but with ECX.

JMP: Jump

JMP imm ; E9 rw/rd [8086]
JMP SHORT imm ; EB rb [8086]

```
JMP imm:imm16        ; o16 EA iw iw     [8086]
JMP imm:imm32        ; o32 EA id iw     [386]
JMP FAR mem          ; o16 FF /5        [8086]
JMP FAR mem32        ; o32 FF /5        [386]
JMP r/m16            ; o16 FF /4        [8086]
JMP r/m32            ; o32 FF /4        [386]
```

JMP jumps to a given address. The address may be specified as an absolute segment and offset, or as a relative jump within the current segment.

JMP SHORT imm has a maximum range of 128 bytes, since the displacement is specified as only 8 bits, but takes up less code space. NASM does not choose when to generate JMP SHORT for you: you must explicitly code SHORT every time you want a short jump.

You can choose between the two immediate far jump forms (JMP imm:imm) by the use of the WORD and DWORD keywords: JMP WORD 0x1234:0x5678) or JMP DWORD 0x1234:0x56789abc .

The JMP FAR mem forms execute a far jump by loading the destination address out of memory.

The address loaded consists of 16 or 32 bits of offset (depending on the operand size), and 16 bits of segment. The operand size may be overridden using JMP WORD FAR mem or JMP DWORD FAR mem .

The JMP r/m forms execute a near jump (within the same segment), loading the destination address out of memory or out of a register. The keyword NEAR may be specified, for clarity, in these forms, but is not necessary. Again, operand size can be overridden using JMP WORD mem or JMP DWORD mem .

As a convenience, NASM does not require you to jump to a far symbol by coding the cumbersome JMP SEG routine:routine , but instead allows the easier synonym JMP FAR routine.

The `CALL r/m` forms given above are near calls; NASM will accept the `NEAR` keyword (e.g. `CALL NEAR [address]`), even though it is not strictly necessary.

`LEA`: Load Effective Address

```
LEA reg16,mem                ; o16 8D /r          [8086]
LEA reg32,mem                ; o32 8D /r          [386]
```

`LEA`, despite its syntax, does not access memory. It calculates the effective address specified by its second operand as if it were going to load or store data from it, but instead it stores the calculated address into the register specified by its first operand. This can be used to perform quite complex calculations (e.g. `LEA EAX,[EBX+ECX*4+100]`) in one instruction.

`LEA`, despite being a purely arithmetic instruction which accesses no memory, still requires square brackets around its second operand, as if it were a memory reference.

The size of the calculation is the current *address* size, and the size that the result is stored as is the current *operand* size. If the address and operand size are not the same, then if the addressing mode was 32–bits, the low 16–bits are stored, and if the address was 16–bits, it is zero–extended to 32–bits before storing.

`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.

LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ: Loop with Counter

```
LOOP imm                    ; E2 rb              [8086]
LOOP imm,CX                 ; a16 E2 rb          [8086]
LOOP imm,ECX                ; a32 E2 rb          [386]
LOOPE imm                   ; E1 rb              [8086]
LOOPE imm,CX                ; a16 E1 rb          [8086]
LOOPE imm,ECX               ; a32 E1 rb          [386]
LOOPZ imm                   ; E1 rb              [8086]
LOOPZ imm,CX                ; a16 E1 rb          [8086]
LOOPZ imm,ECX               ; a32 E1 rb          [386]
LOOPNE imm                  ; E0 rb              [8086]
LOOPNE imm,CX               ; a16 E0 rb          [8086]
LOOPNE imm,ECX              ; a32 E0 rb          [386]
LOOPNZ imm                  ; E0 rb              [8086]
LOOPNZ imm,CX               ; a16 E0 rb          [8086]
LOOPNZ imm,ECX              ; a32 E0 rb          [386]
```

`LOOP` decrements its counter register (either `CX` or `ECX` – if one is not specified explicitly, the `BITS` setting dictates which is used) by one, and if the counter does not become zero as a result of this operation, it jumps to the given label. The jump has a range of 128 bytes.
`LOOPE` (or its synonym `LOOPZ`) adds the additional condition that it only jumps if the counter is nonzero *and* the zero flag is set. Similarly, `LOOPNE` (and `LOOPNZ`) jumps only if the counter is nonzero and the zero flag is clear.

MOV: Move Data

```
MOV r/m8,reg8               ; 88 /r              [8086]
MOV r/m16,reg16             ; o16 89 /r          [8086]
MOV r/m32,reg32             ; o32 89 /r          [386]
MOV reg8,r/m8               ; 8A /r              [8086]
MOV reg16,r/m16             ; o16 8B /r          [8086]
MOV reg32,r/m32             ; o32 8B /r          [386]
MOV reg8,imm8               ; B0+r ib            [8086]
MOV reg16,imm16             ; o16 B8+r iw        [8086]
MOV reg32,imm32             ; o32 B8+r id        [386]
MOV r/m8,imm8               ; C6 /0 ib           [8086]
MOV r/m16,imm16             ; o16 C7 /0 iw       [8086]
MOV r/m32,imm32             ; o32 C7 /0 id       [386]
MOV AL,memoffs8             ; A0 ow/od           [8086]
MOV AX,memoffs16            ; o16 A1 ow/od       [8086]
MOV EAX,memoffs32           ; o32 A1 ow/od       [386]
MOV memoffs8,AL             ; A2 ow/od           [8086]
MOV memoffs16,AX            ; o16 A3 ow/od       [8086]
MOV memoffs32,EAX           ; o32 A3 ow/od       [386]
MOV r/m16,segreg            ; o16 8C /r          [8086]
MOV r/m32,segreg            ; o32 8C /r          [386]
MOV segreg,r/m16            ; o16 8E /r          [8086]
MOV segreg,r/m32            ; o32 8E /r          [386]
MOV reg32,CR0/2/3/4         ; 0F 20 /r           [386]
MOV reg32,DR0/1/2/3/6/7     ; 0F 21 /r           [386]
MOV reg32,TR3/4/5/6/7       ; 0F 24 /r           [386]
```

```
MOV CR0/2/3/4,reg32            ; 0F 22 /r        [386]
MOV DR0/1/2/3/6/7,reg32        ; 0F 23 /r        [386]
MOV TR3/4/5/6/7,reg32          ; 0F 26 /r        [386]
```

`MOV` copies the contents of its source (second) operand into its destination (first) operand.
In all forms of the `MOV` instruction, the two operands are the same size, except for moving between a segment register and an `r/m32` operand. These instructions are treated exactly like the corresponding 16–bit equivalent (so that, for example, `MOV DS,EAX` functions identically to `MOV DS,AX` but saves a prefix when in 32–bit mode), except that when a segment register is moved into a 32–bit destination, the top two bytes of the result are undefined.
`MOV` may not use `CS` as a destination.
`CR4` is only a supported register on the Pentium and above.
Test registers are supported on 386/486 processors and on some non–Intel Pentium class processors.

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.

MUL: Unsigned Integer Multiply

```
MUL r/m8                       ; F6 /4           [8086]
MUL r/m16                      ; o16 F7 /4       [8086]
MUL r/m32                      ; o32 F7 /4       [386]
```

`MUL` performs unsigned integer multiplication. The other operand to the multiplication, and the destination operand, are implicit, in the following way:
For `MUL r/m8`, `AL` is multiplied by the given operand; the product is stored in `AX`.
For `MUL r/m16`, `AX` is multiplied by the given operand; the product is stored in `DX:AX`
For `MUL r/m32`, `EAX` is multiplied by the given operand; the product is stored in `EDX:EAX`.
Signed integer multiplication is performed by the `IMUL` instruction.

NEG, NOT: Two's and One's Complement

```
NEG r/m8                       ; F6 /3           [8086]
NEG r/m16                      ; o16 F7 /3       [8086]
NEG r/m32                      ; o32 F7 /3       [386]
NOT r/m8                       ; F6 /2           [8086]
NOT r/m16                      ; o16 F7 /2       [8086]
```

```
NOT r/m32                     ; o32 F7 /2              [386]
```

`NEG` replaces the contents of its operand by the two's complement negation (invert all the bits and then add one) of the original value. `NOT`, similarly, performs one's complement (inverts all the bits).

`NOP`: No Operation

```
NOP                           ; 90                     [8086]
```

`NOP` performs no operation. Its opcode is the same as that generated by `XCHG AX,AX` or `XCHG EAX,EAX` (depending on the processor mode; see section B.4.333).

`OR`: Bitwise OR

```
OR r/m8,reg8                  ; 08 /r                  [8086]
OR r/m16,reg16                ; o16 09 /r              [8086]
OR r/m32,reg32                ; o32 09 /r              [386]
OR reg8,r/m8                  ; 0A /r                  [8086]
OR reg16,r/m16                ; o16 0B /r              [8086]
OR reg32,r/m32                ; o32 0B /r              [386]
OR r/m8,imm8                  ; 80 /1 ib               [8086]
OR r/m16,imm16                ; o16 81 /1 iw           [8086]
OR r/m32,imm32                ; o32 81 /1 id           [386]
OR r/m16,imm8                 ; o16 83 /1 ib           [8086]
OR r/m32,imm8                 ; o32 83 /1 ib           [386]
OR AL,imm8                    ; 0C ib                  [8086]
OR AX,imm16                   ; o16 0D iw              [8086]
OR EAX,imm32                  ; o32 0D id              [386]
```

`OR` performs a bitwise OR operation between its two operands (i.e. each bit of the result is 1 if and only if at least one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.
In the forms with an 8–bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign–extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

`OUT`: Output Data to I/O Port

```
OUT imm8,AL                   ; E6 ib                  [8086]
OUT imm8,AX                   ; o16 E7 ib              [8086]
OUT imm8,EAX                  ; o32 E7 ib              [386]
OUT DX,AL                     ; EE                     [8086]
OUT DX,AX                     ; o16 EF                 [8086]
OUT DX,EAX                    ; o32 EF                 [386]
```

`OUT` writes the contents of the given source register to the specified I/O port. The port number may be specified as an immediate value if it is between 0 and 255, and otherwise must be stored in `DX`.
See also `IN`.

`OUTSB` , `OUTSW` , `OUTSD` : Output String to I/O Port

```
OUTSB                         ; 6E                     [186]
OUTSW                         ; o16 6F                 [186]
OUTSD                         ; o32 6F                 [386]
```

`OUTSB` loads a byte from `[DS:SI]` or `[DS:ESI]` and writes it to the I/O port specified in `DX`. 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 outsb`).
`OUTSW` and `OUTSD` work in the same way, but they output 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.

POP: Pop Data from Stack

```
POP reg16                      ; o16 58+r              [8086]
POP reg32                      ; o32 58+r              [386]
POP r/m16                      ; o16 8F /0             [8086]
POP r/m32                      ; o32 8F /0             [386]
POP CS                         ; 0F                  [8086,UNDOC]
POP DS                         ; 1F                    [8086]
POP ES                         ; 07                    [8086]
POP SS                         ; 17                    [8086]
POP FS                         ; 0F A1                 [386]
POP GS                         ; 0F A9                 [386]
```

`POP` loads a value from the stack (from `[SS:SP]` or `[SS:ESP]`) and then increments the stack pointer.
The address–size attribute of the instruction determines whether `SP` or `ESP` is used as the stack pointer: to deliberately override the default given by the `BITS` setting, you can use an `a16` or `a32` prefix.
The operand–size attribute of the instruction determines whether the stack pointer is incremented by 2 or 4: this means that segment register pops in `BITS 32` mode will pop 4 bytes off the stack and discard the upper two of them. If you need to override that, you can use an `o16` or `o32` prefix.
The above opcode listings give two forms for general–purpose register pop instructions: for example, `POP BX` has the two forms `5B` and `8F C3` . NASM will always generate the shorter form when given `POP BX` . NDISASM will disassemble both.
`POP CS` is not a documented instruction, and is not supported on any processor above the 8086 (since they use `0Fh` as an opcode prefix for instruction set extensions). However, at least some 8086 processors do support it, and so NASM generates it for completeness.

POPAx : Pop All General–Purpose Registers

```
POPA                           ; 61                    [186]
POPAW                          ; o16 61                [186]
POPAD                          ; o32 61                [386]
```

`POPAW` pops a word from the stack into each of, successively, `DI`, `SI`, `BP`, nothing (it discards a word from the stack which was a placeholder for `SP`), `BX`, `DX`, `CX` and `AX`. It is intended to reverse the operation of `PUSHAW` (see section B.4.264), but it ignores the value for `SP` that was pushed on the stack by `PUSHAW` .
`POPAD` pops twice as much data, and places the results in `EDI`, `ESI`, `EBP`, nothing (placeholder
for `ESP`), `EBX`, `EDX`, `ECX` and `EAX`. It reverses the operation of `PUSHAD` .
`POPA` is an alias mnemonic for either `POPAW` or `POPAD` , depending on the current `BITS` setting.
Note that the registers are popped in reverse order of their numeric values in opcodes.

POPFx : Pop Flags Register

```
POPF                           ; 9D                    [8086]
POPFW                          ; o16 9D                [8086]
```

```
POPFD                         ; o32 9D                [386]
```

`POPFW` pops a word from the stack and stores it in the bottom 16 bits of the flags register (or the whole flags register, on processors below a 386).
`POPFD` pops a doubleword and stores it in the entire flags register.
`POPF` is an alias mnemonic for either `POPFW` or `POPFD`, depending on the current `BITS` setting.
See also `PUSHF`

`PUSH`: Push Data on Stack

```
PUSH reg16                    ; o16 50+r              [8086]
PUSH reg32                    ; o32 50+r              [386]
PUSH r/m16                    ; o16 FF /6             [8086]
PUSH r/m32                    ; o32 FF /6             [386]
PUSH CS                       ; 0E                    [8086]
PUSH DS                       ; 1E                    [8086]
PUSH ES                       ; 06                    [8086]
PUSH SS                       ; 16                    [8086]
PUSH FS                       ; 0F A0                 [386]
PUSH GS                       ; 0F A8                 [386]
PUSH imm8                     ; 6A ib                 [186]
PUSH imm16                    ; o16 68 iw             [186]
PUSH imm32                    ; o32 68 id             [386]
```

`PUSH` decrements the stack pointer (`SP` or `ESP`) by 2 or 4, and then stores the given value at `[SS:SP]` or `[SS:ESP]`.
The address–size attribute of the instruction determines whether `SP` or `ESP` is used as the stack pointer: to deliberately override the default given by the `BITS` setting, you can use an `a16` or `a32` prefix.
The operand–size attribute of the instruction determines whether the stack pointer is decremented by 2 or 4: this means that segment register pushes in `BITS 32` mode will push 4 bytes on the stack, of which the upper two are undefined. If you need to override that, you can use an `o16` or `o32` prefix.
The above opcode listings give two forms for general–purpose register push instructions: for example, `PUSH BX` has the two forms `53` and `FF F3`. NASM will always generate the shorter form when given `PUSH BX`. NDISASM will disassemble both.
Unlike the undocumented and barely supported `POP CS`, `PUSH CS` is a perfectly valid and sensible instruction, supported on all processors.
The instruction `PUSH SP` may be used to distinguish an 8086 from later processors: on an 8086, the value of `SP` stored is the value it has *after* the push instruction, whereas on later processors it is the value *before* the push instruction.

`PUSHAx` : Push All General–Purpose Registers

```
PUSHA                         ; 60                    [186]
PUSHAD                        ; o32 60                [386]
PUSHAW                        ; o16 60                [186]
```

`PUSHAW` pushes, in succession, `AX`, `CX`, `DX`, `BX`, `SP`, `BP`, `SI` and `DI` on the stack, decrementing the stack pointer by a total of 16.
`PUSHAD` pushes, in succession, `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI` and `EDI` on the stack, decrementing the stack pointer by a total of 32.
In both cases, the value of `SP` or `ESP` pushed is its *original* value, as it had before the instruction was executed.
`PUSHA` is an alias mnemonic for either `PUSHAW` or `PUSHAD`, depending on the current `BITS` setting.
Note that the registers are pushed in order of their numeric values in opcodes.
See also `POPA`.

PUSHFx : Push Flags Register

```
PUSHF                          ; 9C                 [8086]
PUSHFD                         ; o32 9C             [386]
PUSHFW                         ; o16 9C             [8086]
```

`PUSHFW` pushes the bottom 16 bits of the flags register (or the whole flags register, on processors below a 386) onto the stack.
`PUSHFD` pushes the entire flags register onto the stack.
`PUSHF` is an alias mnemonic for either `PUSHFW` or `PUSHFD` , depending on the current `BITS` setting.
See also `POPF`

RET, **RETF**, **RETN**: Return from Procedure Call

```
RET                            ; C3                 [8086]
RET imm16                      ; C2 iw              [8086]
RETF                           ; CB                 [8086]
RETF imm16                     ; CA iw              [8086]
RETN                           ; C3                 [8086]
RETN imm16                     ; C2 iw              [8086]
```

`RET`, and its exact synonym `RETN` , pop `IP` or `EIP` from the stack and transfer control to the new address. Optionally, if a numeric second operand is provided, they increment the stack pointer by a further `imm16` bytes after popping the return address.
`RETF` executes a far return: after popping `IP/EIP`, it then pops `CS`, and *then* increments the stack pointer by the optional argument if present.

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`.

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`.

`SBB`: Subtract with Borrow

```
SBB r/m8,reg8                  ; 18 /r              [8086]
SBB r/m16,reg16                ; o16 19 /r          [8086]
SBB r/m32,reg32                ; o32 19 /r          [386]
SBB reg8,r/m8                  ; 1A /r              [8086]
SBB reg16,r/m16                ; o16 1B /r          [8086]
SBB reg32,r/m32                ; o32 1B /r          [386]
SBB r/m8,imm8                  ; 80 /3 ib           [8086]
SBB r/m16,imm16                ; o16 81 /3 iw       [8086]
SBB r/m32,imm32                ; o32 81 /3 id       [386]
SBB r/m16,imm8                 ; o16 83 /3 ib       [8086]
SBB r/m32,imm8                 ; o32 83 /3 ib       [386]
SBB AL,imm8                    ; 1C ib              [8086]
SBB AX,imm16                   ; o16 1D iw          [8086]
SBB EAX,imm32                  ; o32 1D id          [386]
```

`SBB` performs integer subtraction: it subtracts its second operand, plus the value of the carry flag, from its first, and leaves the result in its destination (first) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent `SBB` instruction.
In the forms with an 8–bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign–extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.
To subtract one number from another without also subtracting the contents of the carry flag, use `SUB`.

`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.

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`.

STC, STD, STI: Set Flags

```
STC                           ; F9                    [8086]
STD                           ; FD                    [8086]
STI                           ; FB                    [8086]
```

These instructions set various flags. `STC` sets the carry flag; `STD` sets the direction flag; and `STI` sets the interrupt flag (thus enabling interrupts).
To clear the carry, direction, or interrupt flags, use the `CLC`, `CLD` and `CLI` instructions. To invert the carry flag, use `CMC`

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.

SUB: Subtract Integers

```
SUB r/m8,reg8                    ; 28 /r                  [8086]
SUB r/m16,reg16                  ; o16 29 /r              [8086]
SUB r/m32,reg32                  ; o32 29 /r              [386]
SUB reg8,r/m8                    ; 2A /r                  [8086]
SUB reg16,r/m16                  ; o16 2B /r              [8086]
SUB reg32,r/m32                  ; o32 2B /r              [386]
SUB r/m8,imm8                    ; 80 /5 ib               [8086]
SUB r/m16,imm16                  ; o16 81 /5 iw           [8086]
SUB r/m32,imm32                  ; o32 81 /5 id           [386]
SUB r/m16,imm8                   ; o16 83 /5 ib           [8086]
SUB r/m32,imm8                   ; o32 83 /5 ib           [386]
SUB AL,imm8                      ; 2C ib                  [8086]
SUB AX,imm16                     ; o16 2D iw              [8086]
SUB EAX,imm32                    ; o32 2D id              [386]
```

`SUB` performs integer subtraction: it subtracts its second operand from its first, and leaves the result in its destination (first) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent `SBB` instruction.
In the forms with an 8–bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign–extended to the length of the first operand. In t`BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

TEST: Test Bits (notional bitwise AND)

```
TEST r/m8,reg8                   ; 84 /r                  [8086]
TEST r/m16,reg16                 ; o16 85 /r              [8086]
TEST r/m32,reg32                 ; o32 85 /r              [386]
TEST r/m8,imm8                   ; F6 /0 ib               [8086]
TEST r/m16,imm16                 ; o16 F7 /0 iw           [8086]
TEST r/m32,imm32                 ; o32 F7 /0 id           [386]
TEST AL,imm8                     ; A8 ib                  [8086]
TEST AX,imm16                    ; o16 A9 iw              [8086]
TEST EAX,imm32                   ; o32 A9 id              [386]
```

`TEST` performs a 'mental' bitwise AND of its two operands, and affects the flags as if the operation had taken place, but does not store the result of the operation anywhere.

XCHG: Exchange

```
XCHG reg8,r/m8                   ; 86 /r                  [8086]
XCHG reg16,r/m8                  ; o16 87 /r              [8086]
XCHG reg32,r/m32                 ; o32 87 /r              [386]
XCHG r/m8,reg8                   ; 86 /r                  [8086]
XCHG r/m16,reg16                 ; o16 87 /r              [8086]
XCHG r/m32,reg32                 ; o32 87 /r              [386]
XCHG AX,reg16                    ; o16 90+r               [8086]
XCHG EAX,reg32                   ; o32 90+r               [386]
```

```
XCHG reg16,AX                ; o16 90+r            [8086]
XCHG reg32,EAX               ; o32 90+r            [386]
```

`XCHG` exchanges the values in its two operands. It can be used with a `LOCK` prefix for purposes of multi–processor synchronisation.
`XCHG AX,AX` or `XCHG EAX,EAX` (depending on the `BITS` setting) generates the opcode `90h`, and so is a synonym for `NOP`.

`XLATB` : Translate Byte in Lookup Table

```
XLAT                         ; D7                  [8086]
XLATB                        ; D7                  [8086]
```

`XLATB` adds the value in `AL`, treated as an unsigned byte, to `BX` or `EBX`, and loads the byte from the
resulting address (in the segment specified by `DS`) back into `AL`.
The base register used is `BX` if the address size is 16 bits, and `EBX` 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 `[BX+AL]` or `[EBX+AL]` can be overridden by using a
segment register name as a prefix (for example, `es xlatb`).

`XOR`: Bitwise Exclusive OR

```
XOR r/m8,reg8                ; 30 /r               [8086]
XOR r/m16,reg16              ; o16 31 /r           [8086]
XOR r/m32,reg32              ; o32 31 /r           [386]
XOR reg8,r/m8                ; 32 /r               [8086]
XOR reg16,r/m16              ; o16 33 /r           [8086]
XOR reg32,r/m32              ; o32 33 /r           [386]
XOR r/m8,imm8                ; 80 /6 ib            [8086]
XOR r/m16,imm16              ; o16 81 /6 iw        [8086]
XOR r/m32,imm32              ; o32 81 /6 id        [386]
XOR r/m16,imm8               ; o16 83 /6 ib        [8086]
XOR r/m32,imm8               ; o32 83 /6 ib        [386]
XOR AL,imm8                  ; 34 ib               [8086]
XOR AX,imm16                 ; o16 35 iw           [8086]
XOR EAX,imm32                ; o32 35 id           [386]
```

`XOR` performs a bitwise XOR operation between its two operands (i.e. each bit of the result is 1 if and only if exactly one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.
In the forms with an 8–bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign–extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.
The `MMX` instruction `PXOR` (see section B.4.266) performs the same operation on the 64–bit `MMX` registers.