CMSC 313
COMPUTER ORGANIZATION
&
ASSEMBLY LANGUAGE
PROGRAMMING

LECTURE 05, FALL 2012

TOPICS TODAY

- Project Policy & Academic Integrity
- Project 1
- Conditional Jump Instructions
- Using Jump Instructions
- Logical (bit manipulation) Instructions

CONDITIONAL JUMPS

Branching Instructions

- JMP = unconditional jump
- Conditional jumps use the flags to decide whether to jump to the given label or to continue.
- The flags were modified by previous arithmetic instructions or by a compare (CMP) instruction.
- The instruction

```
CMP     op1, op2
```

computes the unsigned and two's complement subtraction op1 - op2 and modifies the flags. The contents of op1 are not affected.

Example of CMP instruction

- Suppose AL contains 254. After the instruction:

```
CMP     AL, 17
```

CF = 0, OF = 0, SF = 1 and ZF = 0.

- A JA (jump above) instruction would jump.
- A JG (jump greater than) instruction wouldn't jump.
- Both signed and unsigned comparisons use the same CMP instruction.
- Signed and unsigned jump instructions interpret the flags differently.

More Conditional Jumps

- **Uses flags to determine whether to jump**
  - Example: JAE (jump above or equal) jumps when the Carry Flag = 0

```
CMP     EAX, 1492
JAE     OceanBlue
```

- **Unsigned vs signed jumps**
  - Example: use JAE for unsigned data JGE (greater than or equal) for signed data

```
CMP     EAX, 1492
JAE     OceanBlue
```

```
CMP     EAX,  -42
JGE     Somewhere
```

Table 7-4. Conditional Jump Instructions

Instruction Mnemonic	Condition (Flag States)	Description
Unsigned Conditional Jumps		
JA/JNBE	(CF or ZF)=0	Above/not below or equal
JAE/JNB	CF=0	Above or equal/not below
JB/JNAE	CF=1	Below/not above or equal
JBE/JNA	(CF or ZF)=1	Below or equal/not above
JC	CF=1	Carry
JE/JZ	ZF=1	Equal/zero
JNC	CF=0	Not carry
JNE/JNZ	ZF=0	Not equal/not zero
JNP/JPO	PF=0	Not parity/parity odd
JP/JPE	PF=1	Parity/parity even
JCXZ	CX=0	Register CX is zero
JECXZ	ECX=0	Register ECX is zero
Signed Conditional Jumps		
JG/JNLE	((SF xor OF) or ZF) =0	Greater/not less or equal
JGE/JNL	(SF xor OF)=0	Greater or equal/not less
JL/JNGE	(SF xor OF)=1	Less/not greater or equal
JLE/JNG	((SF xor OF) or ZF)=1	Less or equal/not greater
JNO	OF=0	Not overflow
JNS	SF=0	Not sign (non-negative)
JO	OF=1	Overflow
JS	SF=1	Sign (negative)

J*cc*—Jump if Condition Is Met

Opcode	Instruction	Description
77 *cb*	JA *rel8*	Jump short if above (CF=0 and ZF=0)
73 *cb*	JAE *rel8*	Jump short if above or equal (CF=0)
72 *cb*	JB *rel8*	Jump short if below (CF=1)
76 *cb*	JBE *rel8*	Jump short if below or equal (CF=1 or ZF=1)
72 *cb*	JC *rel8*	Jump short if carry (CF=1)
E3 *cb*	JCXZ *rel8*	Jump short if CX register is 0
E3 *cb*	JECXZ *rel8*	Jump short if ECX register is 0
74 *cb*	JE *rel8*	Jump short if equal (ZF=1)
7F *cb*	JG *rel8*	Jump short if greater (ZF=0 and SF=OF)
7D *cb*	JGE *rel8*	Jump short if greater or equal (SF=OF)
7C *cb*	JL *rel8*	Jump short if less (SF<>OF)
7E *cb*	JLE *rel8*	Jump short if less or equal (ZF=1 or SF<>OF)
76 *cb*	JNA *rel8*	Jump short if not above (CF=1 or ZF=1)
72 *cb*	JNAE *rel8*	Jump short if not above or equal (CF=1)
73 *cb*	JNB *rel8*	Jump short if not below (CF=0)
77 *cb*	JNBE *rel8*	Jump short if not below or equal (CF=0 and ZF=0)
73 *cb*	JNC *rel8*	Jump short if not carry (CF=0)
75 *cb*	JNE *rel8*	Jump short if not equal (ZF=0)
7E *cb*	JNG *rel8*	Jump short if not greater (ZF=1 or SF<>OF)
7C *cb*	JNGE *rel8*	Jump short if not greater or equal (SF<>OF)
7D *cb*	JNL *rel8*	Jump short if not less (SF=OF)
7F *cb*	JNLE *rel8*	Jump short if not less or equal (ZF=0 and SF=OF)
71 *cb*	JNO *rel8*	Jump short if not overflow (OF=0)
7B *cb*	JNP *rel8*	Jump short if not parity (PF=0)
79 *cb*	JNS *rel8*	Jump short if not sign (SF=0)
75 *cb*	JNZ *rel8*	Jump short if not zero (ZF=0)
70 *cb*	JO *rel8*	Jump short if overflow (OF=1)
7A *cb*	JP *rel8*	Jump short if parity (PF=1)
7A *cb*	JPE *rel8*	Jump short if parity even (PF=1)
7B *cb*	JPO *rel8*	Jump short if parity odd (PF=0)
78 *cb*	JS *rel8*	Jump short if sign (SF=1)
74 *cb*	JZ *rel8*	Jump short if zero (ZF 1)
0F 87 *cw/cd*	JA *rel16/32*	Jump near if above (CF=0 and ZF=0)
0F 83 *cw/cd*	JAE *rel16/32*	Jump near if above or equal (CF=0)
0F 82 *cw/cd*	JB *rel16/32*	Jump near if below (CF=1)
0F 86 *cw/cd*	JBE *rel16/32*	Jump near if below or equal (CF=1 or ZF=1)
0F 82 *cw/cd*	JC *rel16/32*	Jump near if carry (CF=1)
0F 84 *cw/cd*	JE *rel16/32*	Jump near if equal (ZF=1)
0F 84 *cw/cd*	JZ *rel16/32*	Jump near if 0 (ZF=1)
0F 8F *cw/cd*	JG *rel16/32*	Jump near if greater (ZF=0 and SF=OF)

J*cc*—Jump if Condition Is Met (Continued)

Opcode	Instruction	Description
0F 8D *cw/cd*	JGE *rel16/32*	Jump near if greater or equal (SF=OF)
0F 8C *cw/cd*	JL *rel16/32*	Jump near if less (SF<>OF)
0F 8E *cw/cd*	JLE *rel16/32*	Jump near if less or equal (ZF=1 or SF<>OF)
0F 86 *cw/cd*	JNA *rel16/32*	Jump near if not above (CF=1 or ZF=1)
0F 82 *cw/cd*	JNAE *rel16/32*	Jump near if not above or equal (CF=1)
0F 83 *cw/cd*	JNB *rel16/32*	Jump near if not below (CF=0)
0F 87 *cw/cd*	JNBE *rel16/32*	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 *cw/cd*	JNC *rel16/32*	Jump near if not carry (CF=0)
0F 85 *cw/cd*	JNE *rel16/32*	Jump near if not equal (ZF=0)
0F 8E *cw/cd*	JNG *rel16/32*	Jump near if not greater (ZF=1 or SF<>OF)
0F 8C *cw/cd*	JNGE *rel16/32*	Jump near if not greater or equal (SF<>OF)
0F 8D *cw/cd*	JNL *rel16/32*	Jump near if not less (SF=OF)
0F 8F *cw/cd*	JNLE *rel16/32*	Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 *cw/cd*	JNO *rel16/32*	Jump near if not overflow (OF=0)
0F 8B *cw/cd*	JNP *rel16/32*	Jump near if not parity (PF=0)
0F 89 *cw/cd*	JNS *rel16/32*	Jump near if not sign (SF=0)
0F 85 *cw/cd*	JNZ *rel16/32*	Jump near if not zero (ZF=0)
0F 80 *cw/cd*	JO *rel16/32*	Jump near if overflow (OF=1)
0F 8A *cw/cd*	JP *rel16/32*	Jump near if parity (PF=1)
0F 8A *cw/cd*	JPE *rel16/32*	Jump near if parity even (PF=1)
0F 8B *cw/cd*	JPO *rel16/32*	Jump near if parity odd (PF=0)
0F 88 *cw/cd*	JS *rel16/32*	Jump near if sign (SF=1)
0F 84 *cw/cd*	JZ *rel16/32*	Jump near if 0 (ZF=1)

Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the J*cc* instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (*rel8*, *rel16,* or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of –128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits.

J*cc*—Jump if Condition Is Met (Continued)

The conditions for each J*cc* mnemonic are given in the "Description" column of the table on the preceding page. The terms "less" and "greater" are used for comparisons of signed integers and the terms "above" and "below" are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The J*cc* instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the J*cc* instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

```
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

```
JNZ BEYOND;
JMP FARLABEL;
BEYOND:
```

The JECXZ and JCXZ instructions differs from the other J*cc* instructions because they do not check the status flags. Instead they check the contents of the ECX and CX registers, respectively, for 0. Either the CX or ECX register is chosen according to the address-size attribute. These instructions are useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as LOOPNE). They prevent entering the loop when the ECX or CX register is equal to 0, which would cause the loop to execute 2^{32} or 64K times, respectively, instead of zero times.

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

Operation

```
IF condition
    THEN
         EIP     EIP + SignExtend(DEST);
        IF OperandSize     16
            THEN
                EIP     EIP AND 0000FFFFH;
        FI;
            ELSE (* OperandSize = 32 *)
                IF EIP < CS.Base OR EIP > CS.Limit
                    #GP
        FI;
FI;
```

J*cc*—Jump if Condition Is Met (Continued)

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the offset being jumped to is beyond the limits of the CS segment.

Real-Address Mode Exceptions

#GP	If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

Closer look at JGE

- JGE jumps if and only if SF = OF
  - Examples using 8-bit registers. Which of these result in a jump?

1.
```
MOV   AL, 96
CMP   AL, 80
JGE   Somewhere
```

2.
```
MOV   AL, -64
CMP   AL, 80
JGE   Somewhere
```

3.
```
MOV   AL, 64
CMP   AL, -80
JGE   Somewhere
```

4.
```
MOV   AL, 64
CMP   AL, 80
JGE   Somewhere
```

- if OF=0, then use SF to check whether A-B >= 0.
- if OF=1, then do opposite of SF.
- JGE works after a CMP instruction, even when subtracting the operands result in an overflow!

SHORT VS NEAR JUMPS

Short Jumps vs Near Jumps

- **Jumps use relative addressing**
  - Assembler computes an "offset" from address of current instruction
  - Code produced is "relocatable"
- **Short jumps use 8-bit offsets**
  - Target label must be -128 bytes to +127 bytes away
  - ~~Conditional jumps use short jumps by default. To use a near jump:~~

    ~~JGE NEAR Somewhere~~
- **Near jumps use 32-bit offsets**
  - Target label must be -2^{32} to $+2^{32}-1$ bytes away (4 gigabyte range)
  - ~~Unconditional jumps use near jumps by default. To use a short jump.~~

    ~~JMP SHORT Somewhere~~

SHORT JUMPS VS NEAR JUMPS

- **Jumps use relative addressing**
  - assembler computes an *offset* from address of current instruction.
  - produces *relocatable* code
- **SHORT jumps use 8-bit offsets**
  - target label within -128 bytes to +127 bytes
- **NEAR jumps use 32-bit offsets**
  - target label within -2^{32} bytes to $+2^{32}-1$ bytes

SHORT JUMPS VS NEAR JUMPS

- **Some assemblers determine SHORT vs NEAR jumps automatically, but *some do not.***

- **explicitly specify SHORT jumps**

```
jmp     SHORT somewhere
```

- **explicitly specify NEAR jumps**

```
jge     NEAR somewhere
```

```
; File: jmp.asm
;
; Demonstrating near and short jumps
;

        section .text
        global _start

_start: nop

        ; initialize

start:  mov     eax, 17         ; eax := 17
        cmp     eax, 42         ; 17 - 42 is ...

        jge     exit            ; exit if 17 >= 42
        jge     short exit
        jge     near exit

        jmp     exit
        jmp     short exit
        jmp     near exit

exit:   mov     ebx, 0          ; exit code, 0=normal
        mov     eax, 1          ; Exit.
        int     080H            ; Call kernel.
```

```
 1                          ; File: jmp.asm
 2                          ;
 3                          ; Demonstrating near and short jumps
 4                          ;
 5
 6                                  section .text
 7                                  global _start
 8
 9 00000000 90              _start: nop
10
11                                  ; initialize
12
13 00000001 B811000000    start:  mov     eax, 17         ; eax := 17
14 00000006 3D2A000000             cmp     eax, 42         ; 17 - 42 is ...
15
16 0000000B 7D14                   jge     exit            ; exit if 17 >= 42
17 0000000D 7D12                   jge     short exit
18 0000000F 0F8D0C000000           jge     near exit
19
20 00000015 E907000000             jmp     exit
21 0000001A EB05                   jmp     short exit
22 0000001C E900000000             jmp     near exit
23
24 00000021 BB00000000    exit:   mov     ebx, 0          ; exit code, 0=normal
25 00000026 B801000000             mov     eax, 1          ; Exit.
26 0000002B CD80                   int     080H            ; Call kernel.
```

USING JUMP INSTRUCTIONS

Converting an if Statement

```
if (x < y) {
   statement block 1 ;
} else {
   statement block 2 ;
}
```

```
        MOV     EAX,[x]
        CMP     EAX,[y]
        JGE     ElsePart
          .                     ; if part
          .                     ; statement block 1
          .
        JMP     Done            ; skip over else part

ElsePart:
          .                     ; else part
          .                     ; statement block 2
          .

Done:
```

Converting a while Loop

```
while(i > 0) {
    statement 1 ;
    statement 2 ;
    …
}
```

```
WhileTop:
        MOV     EAX,[i]
        CMP     EAX, 0
        JLE     Done
          .                     ; statement 1
          .
          .
          .                     ; statement 2
          .
          .
        JMP     WhileTop
Done:
```

BIT MANIPULATION

Logical (bit manipulation) Instructions

- **AND: used to clear bits (store 0 in the bits):**
  - To clear the lower 4 bits of the AL register:

```
AND     AL, F0h          1101 0110
                         1111 0000
                         1101 0000
```

- **OR: used to set bits (store 1 in the bits):**
  - To set the lower 4 bits of the AL register:

```
OR      AL, 0Fh          1101 0110
                         0000 1111
                         1101 1111
```

- **NOT: flip all the bits**
- **Shift and Rotate instructions move bits around**

AND—Logical AND

Opcode	Instruction	Description
24 *ib*	AND AL,*imm8*	AL AND *imm8*
25 *iw*	AND AX,*imm16*	AX AND i*mm16*
25 *id*	AND EAX,*imm32*	EAX AND *imm32*
80 /4 *ib*	AND *r/m8,imm8*	*r/m8* AND *imm8*
81 /4 *iw*	AND *r/m16,imm16*	*r/m16* AND *imm16*
81 /4 *id*	AND *r/m32,imm32*	*r/m32* AND *imm32*
83 /4 *ib*	AND *r/m16,imm8*	*r/m16* AND *imm8 (sign-extended)*
83 /4 *ib*	AND *r/m32,imm8*	*r/m32* AND *imm8 (sign-extended)*
20 */r*	AND *r/m8,r8*	*r/m8* AND *r8*
21 /*r*	AND *r/m16,r16*	*r/m16* AND *r16*
21 /*r*	AND *r/m32,r32*	*r/m32* AND *r32*
22 /*r*	AND *r8,r/m8*	*r8* AND *r/m8*
23 /*r*	AND *r16,r/m16*	*r16* AND *r/m16*
23 /*r*	AND *r32,r/m32*	*r32* AND *r/m32*

Description

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

```
DEST    DEST AND SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.

OR—Logical Inclusive OR

Opcode	Instruction	Description
0C *ib*	OR AL,*imm8*	AL OR *imm8*
0D *iw*	OR AX,*imm16*	AX OR *imm16*
0D *id*	OR EAX,*imm32*	EAX OR *imm32*
80 /1 *ib*	OR *r/m8,imm8*	*r/m8* OR *imm8*
81 /1 *iw*	OR *r/m16,imm16*	*r/m16* OR *imm16*
81 /1 *id*	OR *r/m32,imm32*	*r/m32* OR *imm32*
83 /1 *ib*	OR *r/m16,imm8*	*r/m16* OR *imm8 (sign-extended)*
83 /1 *ib*	OR *r/m32,imm8*	*r/m32* OR *imm8 (sign-extended)*
08 /*r*	OR *r/m8,r8*	*r/m8* OR *r8*
09 /*r*	OR *r/m16,r16*	*r/m16* OR *r16*
09 /*r*	OR *r/m32,r32*	*r/m32* OR *r32*
0A /*r*	OR *r8,r/m8*	*r8* OR *r/m8*
0B /*r*	OR *r16,r/m16*	*r16* OR *r/m16*
0B /*r*	OR *r32,r/m32*	*r32* OR *r/m32*

Description

Performs a bitwise inclusive OR operation between the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

```
DEST    DEST OR SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0) If the destination operand points to a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

NOT—One's Complement Negation

Opcode	Instruction	Description
F6 /2	NOT *r/m8*	Reverse each bit of *r/m8*
F7 /2	NOT *r/m16*	Reverse each bit of *r/m16*
F7 /2	NOT *r/m32*	Reverse each bit of *r/m32*

Description

Performs a bitwise NOT operation (each 1 is cleared to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

```
DEST    NOT DEST;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

SAL/SAR/SHL/SHR—Shift

Opcode	Instruction	Description
D0 /4	SAL *r/m8*,1	Multiply *r/m8* by 2, once
D2 /4	SAL *r/m8*,CL	Multiply *r/m8* by 2, CL times
C0 /4 *ib*	SAL *r/m8,imm8*	Multiply *r/m8* by 2, *imm8* times
D1 /4	SAL *r/m16*,1	Multiply *r/m16* by 2, once
D3 /4	SAL *r/m16*,CL	Multiply *r/m16* by 2, CL times
C1 /4 *ib*	SAL *r/m16,imm8*	Multiply *r/m16* by 2, *imm8* times
D1 /4	SAL *r/m32*,1	Multiply *r/m32* by 2, once
D3 /4	SAL *r/m32*,CL	Multiply *r/m32* by 2, CL times
C1 /4 *ib*	SAL *r/m32,imm8*	Multiply *r/m32* by 2, *imm8* times
D0 /7	SAR *r/m8*,1	Signed divide* *r/m8* by 2, once
D2 /7	SAR *r/m8*,CL	Signed divide* *r/m8* by 2, CL times
C0 /7 *ib*	SAR *r/m8,imm8*	Signed divide* *r/m8* by 2, *imm8* times
D1 /7	SAR *r/m16*,1	Signed divide* *r/m16* by 2, once
D3 /7	SAR *r/m16*,CL	Signed divide* *r/m16* by 2, CL times
C1 /7 *ib*	SAR *r/m16,imm8*	Signed divide* *r/m16* by 2, *imm8* times
D1 /7	SAR *r/m32*,1	Signed divide* *r/m32* by 2, once
D3 /7	SAR *r/m32*,CL	Signed divide* *r/m32* by 2, CL times
C1 /7 *ib*	SAR *r/m32,imm8*	Signed divide* *r/m32* by 2, *imm8* times
D0 /4	SHL *r/m8*,1	Multiply *r/m8* by 2, once
D2 /4	SHL *r/m8*,CL	Multiply *r/m8* by 2, CL times
C0 /4 *ib*	SHL *r/m8,imm8*	Multiply *r/m8* by 2, *imm8* times
D1 /4	SHL *r/m16*,1	Multiply *r/m16* by 2, once
D3 /4	SHL *r/m16*,CL	Multiply *r/m16* by 2, CL times
C1 /4 *ib*	SHL *r/m16,imm8*	Multiply *r/m16* by 2, *imm8* times
D1 /4	SHL *r/m32*,1	Multiply *r/m32* by 2, once
D3 /4	SHL *r/m32*,CL	Multiply *r/m32* by 2, CL times
C1 /4 *ib*	SHL *r/m32,imm8*	Multiply *r/m32* by 2, *imm8* times
D0 /5	SHR *r/m8*,1	Unsigned divide *r/m8* by 2, once
D2 /5	SHR *r/m8*,CL	Unsigned divide *r/m8* by 2, CL times
C0 /5 *ib*	SHR *r/m8,imm8*	Unsigned divide *r/m8* by 2, *imm8* times
D1 /5	SHR *r/m16*,1	Unsigned divide *r/m16* by 2, once
D3 /5	SHR *r/m16*,CL	Unsigned divide *r/m16* by 2, CL times
C1 /5 *ib*	SHR *r/m16,imm8*	Unsigned divide *r/m16* by 2, *imm8* times
D1 /5	SHR *r/m32*,1	Unsigned divide *r/m32* by 2, once
D3 /5	SHR *r/m32*,CL	Unsigned divide *r/m32* by 2, CL times
C1 /5 *ib*	SHR *r/m32,imm8*	Unsigned divide *r/m32* by 2, *imm8* times

NOTE:

* Not the same form of division as IDIV; rounding is toward negative infinity.

SAL/SAR/SHL/SHR—Shift (Continued)

Description

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or register CL. The count is masked to 5 bits, which limits the count range to 0 to 31. A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value (see Figure 7-9 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the "quotient" of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the "remainder" is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is cleared to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

SAL/SAR/SHL/SHR—Shift (Continued)

IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
tempCOUNT     (COUNT AND 1FH);
tempDEST     DEST;
WHILE (tempCOUNT    0)
DO
    IF instruction is SAL or SHL
        THEN
            CF     MSB(DEST);
        ELSE (* instruction is SAR or SHR *)
            CF     LSB(DEST);
    FI;
    IF instruction is SAL or SHL
        THEN
            DEST     DEST * 2;
        ELSE
            IF instruction is SAR
                THEN
                    DEST     DEST / 2 (*Signed divide, rounding toward negative infinity*);
                ELSE (* instruction is SHR *)
                    DEST     DEST / 2 ; (* Unsigned divide *);
            FI;
    FI;
    tempCOUNT     tempCOUNT – 1;
OD;
(* Determine overflow for the various instructions *)
IF COUNT     1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF     MSB(DEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF     0;
                    ELSE (* instruction is SHR *)
                        OF     MSB(tempDEST);
                FI;
        FI;
```

SAL/SAR/SHL/SHR—Shift (Continued)

```
    ELSE IF COUNT     0
        THEN
            All flags remain unchanged;
        ELSE (* COUNT neither 1 or 0 *)
            OF     undefined;
    FI;
FI;
```

Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see "Description" above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

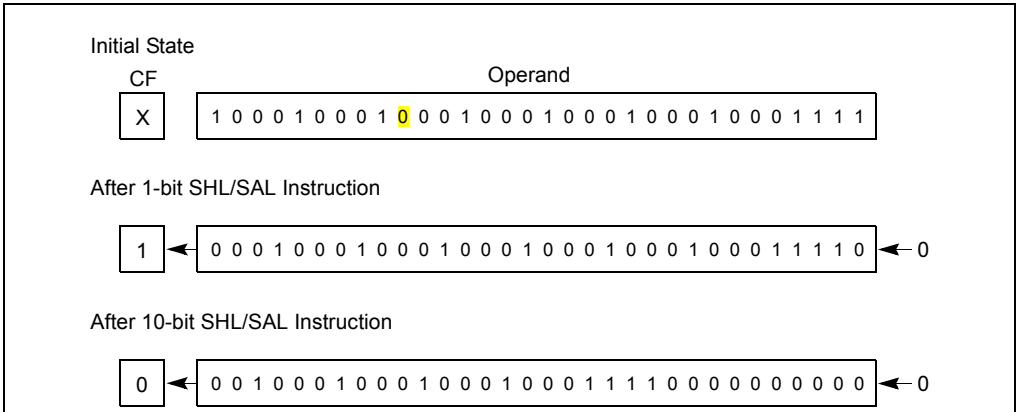


Figure 7-7. SHL/SAL Instruction Operation

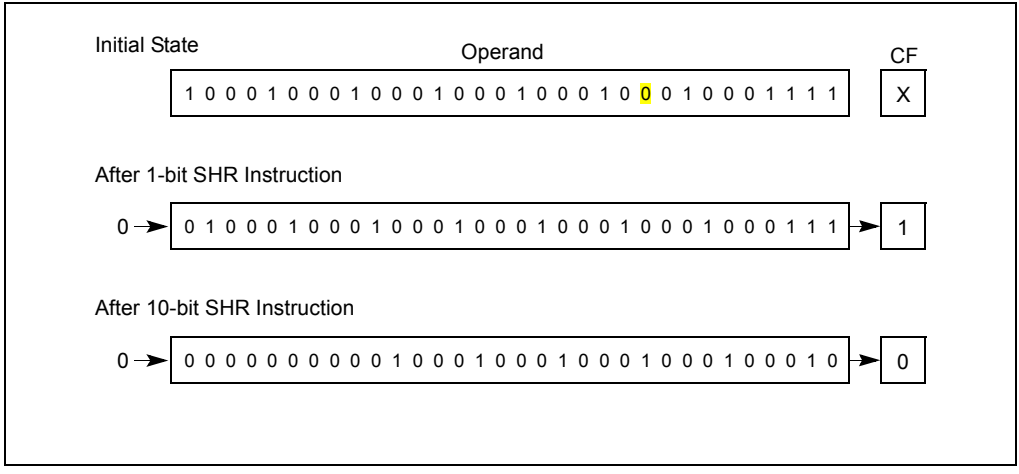


Figure 7-8. SHR Instruction Operation

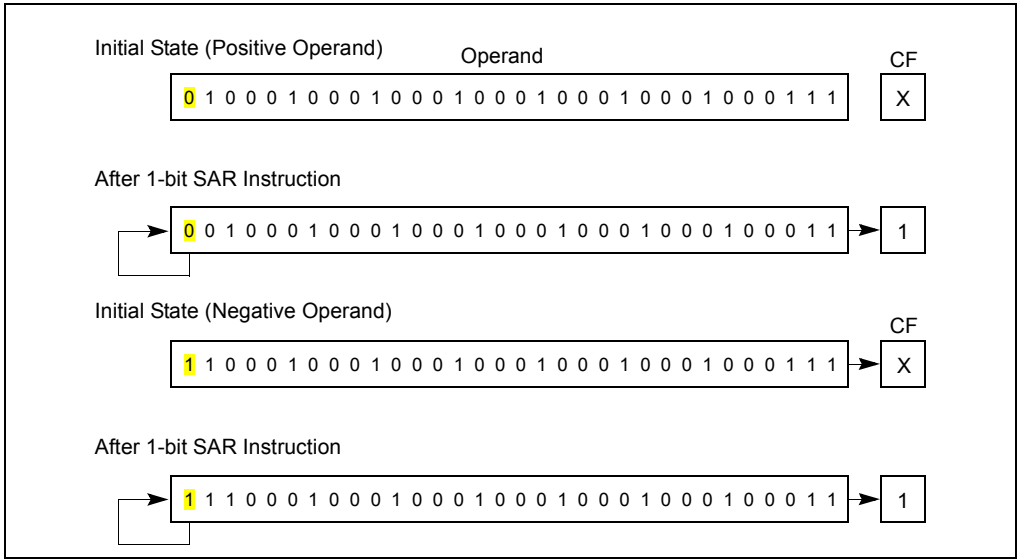


Figure 7-9. SAR Instruction Operation

RCL/RCR/ROL/ROR—Rotate

Opcode	Instruction	Description
D0 /2	RCL *r/m8*, 1	Rotate 9 bits (CF, *r/m8*) left once
D2 /2	RCL *r/m8*, CL	Rotate 9 bits (CF, *r/m8*) left CL times
C0 /2 *ib*	RCL *r/m8, imm8*	Rotate 9 bits (CF, *r/m8*) left *imm8* times
D1 /2	RCL *r/m16*, 1	Rotate 17 bits (CF, *r/m16*) left once
D3 /2	RCL *r/m16*, CL	Rotate 17 bits (CF, *r/m16*) left CL times
C1 /2 *ib*	RCL *r/m16, imm8*	Rotate 17 bits (CF, *r/m16*) left *imm8* times
D1 /2	RCL *r/m32*, 1	Rotate 33 bits (CF, *r/m32*) left once
D3 /2	RCL *r/m32*, CL	Rotate 33 bits (CF, *r/m32*) left CL times
C1 /2 *ib*	RCL *r/m32,i mm8*	Rotate 33 bits (CF, *r/m32*) left *imm8* times
D0 /3	RCR *r/m8*, 1	Rotate 9 bits (CF, *r/m8*) right once
D2 /3	RCR *r/m8*, CL	Rotate 9 bits (CF, *r/m8*) right CL times
C0 /3 *ib*	RCR *r/m8, imm8*	Rotate 9 bits (CF, *r/m8*) right *imm8* times
D1 /3	RCR *r/m16*, 1	Rotate 17 bits (CF, *r/m16*) right once
D3 /3	RCR *r/m16*, CL	Rotate 17 bits (CF, *r/m16*) right CL times
C1 /3 *ib*	RCR *r/m16, imm8*	Rotate 17 bits (CF, *r/m16*) right *imm8* times
D1 /3	RCR *r/m32*, 1	Rotate 33 bits (CF, *r/m32*) right once
D3 /3	RCR *r/m32*, CL	Rotate 33 bits (CF, *r/m32*) right CL times
C1 /3 *ib*	RCR *r/m32, imm8*	Rotate 33 bits (CF, *r/m32*) right *imm8* times
D0 /0	ROL *r/m8*, 1	Rotate 8 bits *r/m8* left once
D2 /0	ROL *r/m8*, CL	Rotate 8 bits *r/m8* left CL times
C0 /0 *ib*	ROL *r/m8, imm8*	Rotate 8 bits *r/m8* left *imm8* times
D1 /0	ROL *r/m16*, 1	Rotate 16 bits *r/m16* left once
D3 /0	ROL *r/m16*, CL	Rotate 16 bits *r/m16* left CL times
C1 /0 *ib*	ROL *r/m16, imm8*	Rotate 16 bits *r/m16* left *imm8* times
D1 /0	ROL *r/m32*, 1	Rotate 32 bits *r/m32* left once
D3 /0	ROL *r/m32*, CL	Rotate 32 bits *r/m32* left CL times
C1 /0 *ib*	ROL *r/m32, imm8*	Rotate 32 bits *r/m32* left *imm8* times
D0 /1	ROR *r/m8*, 1	Rotate 8 bits *r/m8* right once
D2 /1	ROR *r/m8*, CL	Rotate 8 bits *r/m8* right CL times
C0 /1 *ib*	ROR *r/m8, imm8*	Rotate 8 bits *r/m16* right *imm8* times
D1 /1	ROR *r/m16*, 1	Rotate 16 bits *r/m16* right once
D3 /1	ROR *r/m16*, CL	Rotate 16 bits *r/m16* right CL times
C1 /1 *ib*	ROR *r/m16, imm8*	Rotate 16 bits *r/m16* right *imm8* times
D1 /1	ROR *r/m32*, 1	Rotate 32 bits *r/m32* right once
D3 /1	ROR *r/m32*, CL	Rotate 32 bits *r/m32* right CL times
C1 /1 *ib*	ROR *r/m32, imm8*	Rotate 32 bits *r/m32* right *imm8* times

RCL/RCR/ROL/ROR—Rotate (Continued)

Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The processor restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*).

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except that a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
(* RCL and RCR instructions *)
SIZE    OperandSize
CASE (determine count) OF
    SIZE    8:  tempCOUNT    (COUNT AND 1FH) MOD 9;
    SIZE    16: tempCOUNT    (COUNT AND 1FH) MOD 17;
    SIZE    32: tempCOUNT    COUNT AND 1FH;
ESAC;
```

RCL/RCR/ROL/ROR—Rotate (Continued)

```
(* RCL instruction operation *)
WHILE (tempCOUNT    0)
    DO
        tempCF     MSB(DEST);
        DEST     (DEST ∗ 2) + CF;
        CF     tempCF;
        tempCOUNT     tempCOUNT – 1;
    OD;
ELIHW;
IF COUNT     1
    THEN OF     MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
(* RCR instruction operation *)
IF COUNT     1
    THEN OF     MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
WHILE (tempCOUNT    0)
    DO
        tempCF     LSB(SRC);
        DEST     (DEST / 2) + (CF * 2SIZE);
        CF     tempCF;
        tempCOUNT     tempCOUNT – 1;
    OD;
(* ROL and ROR instructions *)
SIZE     OperandSize
CASE (determine count) OF
    SIZE     8:    tempCOUNT     COUNT MOD 8;
    SIZE     16:   tempCOUNT     COUNT MOD 16;
    SIZE     32:   tempCOUNT     COUNT MOD 32;
ESAC;
(* ROL instruction operation *)
WHILE (tempCOUNT    0)
    DO
        tempCF     MSB(DEST);
        DEST     (DEST ∗ 2) + tempCF;
        tempCOUNT     tempCOUNT – 1;
    OD;
ELIHW;
CF     LSB(DEST);
IF COUNT     1
    THEN OF     MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
```

RCL/RCR/ROL/ROR—Rotate (Continued)

```
(* ROR instruction operation *)
WHILE (tempCOUNT    0)
    DO
        tempCF     LSB(SRC);
        DEST     (DEST / 2) + (tempCF ∗ 2^SIZE);
        tempCOUNT     tempCOUNT – 1;
    OD;
ELIHW;
CF     MSB(DEST);
IF COUNT     1
    THEN OF     MSB(DEST) XOR MSB – 1(DEST);
    ELSE OF is undefined;
FI;
```

Flags Affected

The CF flag contains the value of the bit shifted into it. The OF flag is affected only for single-bit rotates (see “Description” above); it is undefined for multi-bit rotates. The SF, ZF, AF, and PF flags are not affected.

Protected Mode Exceptions

#GP(0)	If the source operand is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

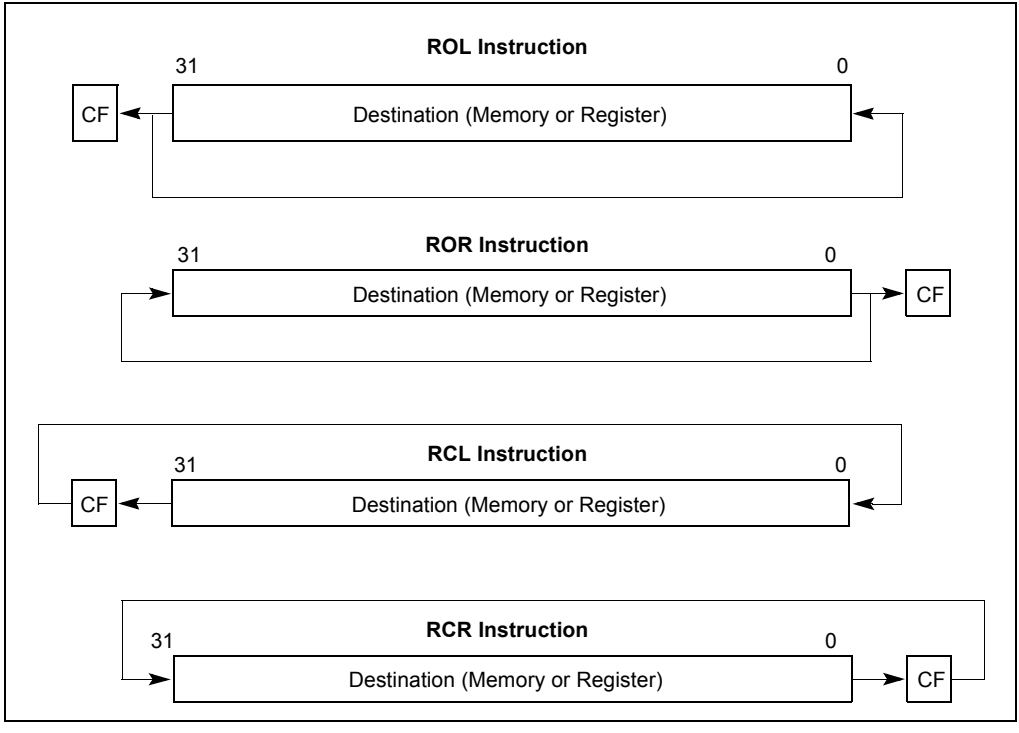
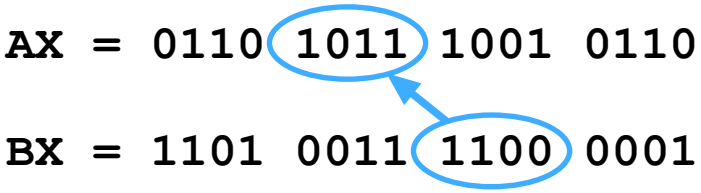


Figure 7-11. ROL, ROR, RCL, and RCR Instruction Operations

Example using AND, OR & SHL

- **Copy bits 4-7 of BX to bits 8-11 of AX**



1. Clear bits 8-11 of AX & all but bits 4-7 of BX using AND instructions

```
AX = 0110 0000 1001 0110        AND    AX, F0FFh
BX = 0000 0000 1100 0000        AND    BX, 00F0h
```

2. Shift bits 4-7 of BX to the desired position using a SHL instruction

```
AX = 0110 0000 1001 0110
BX = 0000 1100 0000 0000        SHL    BX, 4
```

3. "Copy" bits of 4-7 of BX to AX using an OR instruction

```
AX = 0110 1100 1001 0110        OR     AX, BX
BX = 0000 1100 0000 0000
```

NEXT TIME

- More arithmetic operations
- Indexed addressing: [ESI + 4*ECX + 1024]
- Example: a complex i386 instruction

References

- **Some figures and diagrams from *IA-32 Intel Architecture Software Developer's Manual, Vols 1-3***

  <http://developer.intel.com/design/Pentium4/manuals/>