x86 Assembly (cont.)

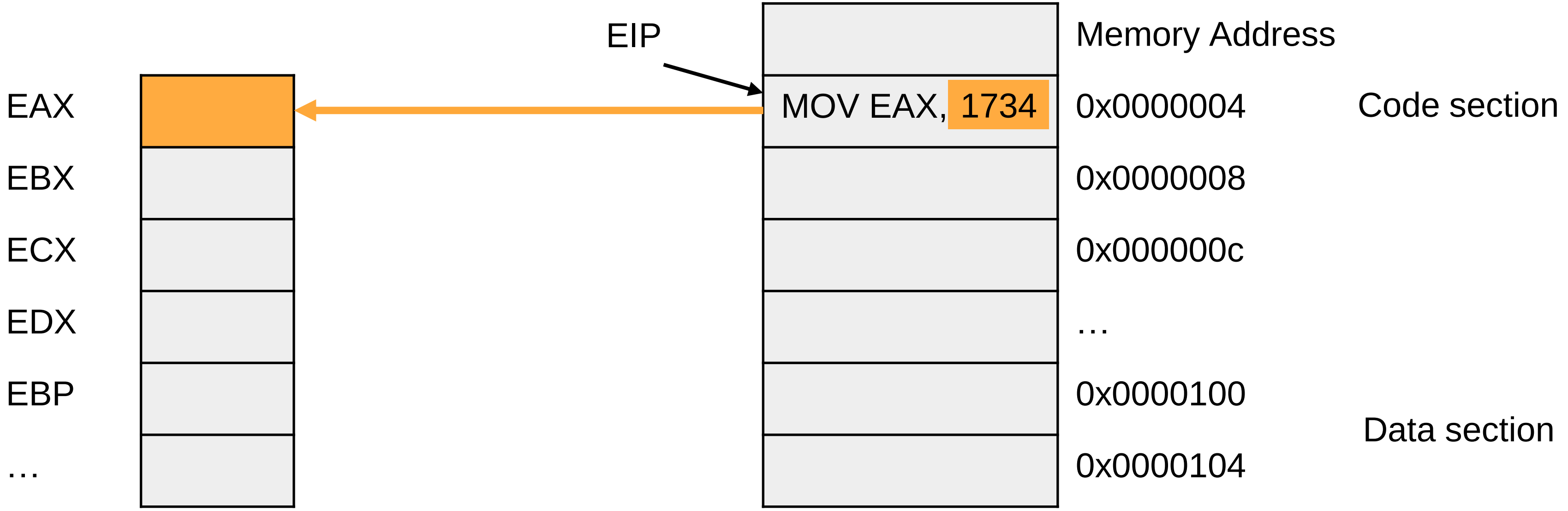
CMSC 313
Raphael Elspas

Addressing Modes

- Addressing modes refers to the way instruction operands are specified. We need to consider the addressing mode for the different instructions
- Modes include:
    - Immediate
    - Direct
    - Indirect
    - Register
    - Register Indirect
    - Displacement
    - Stack

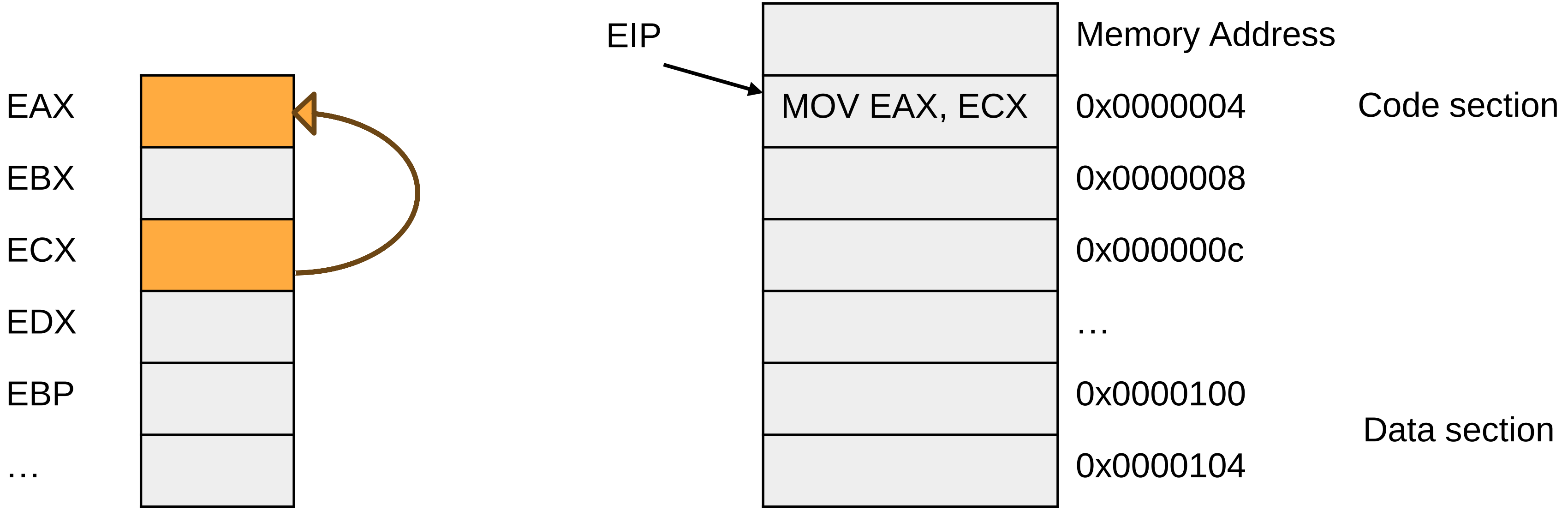
Immediate to register

```
MOV EAX, 1734
```



Register to register

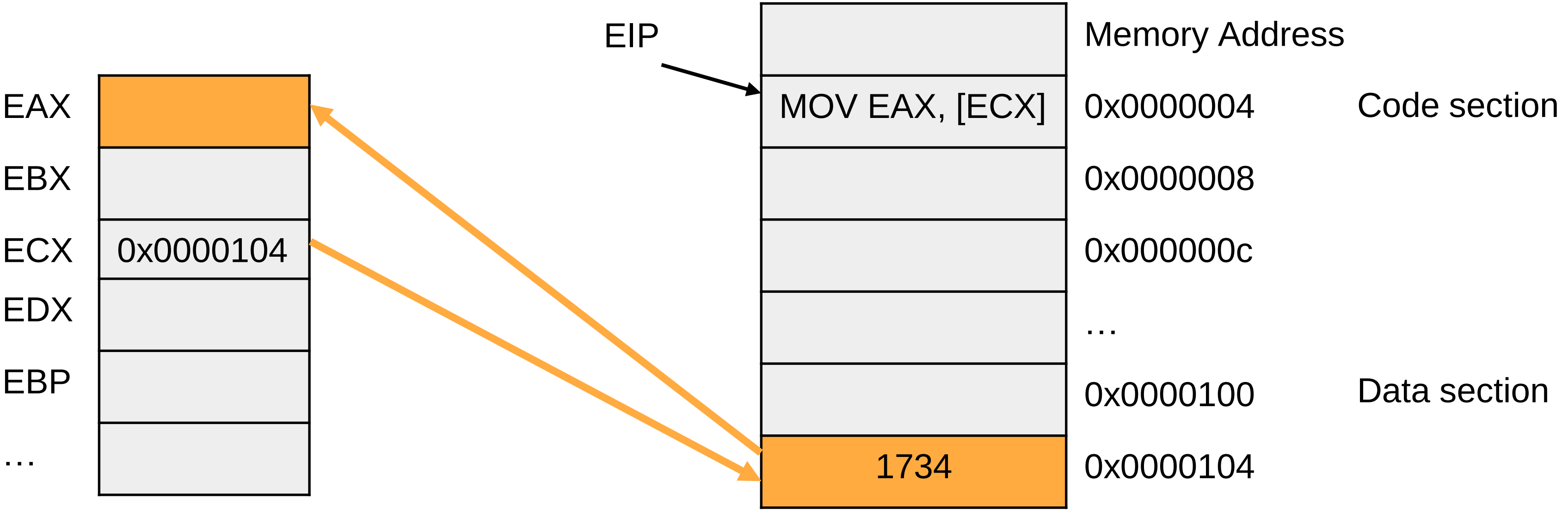
```
MOV EAX, ECX
```



Register indirect to register

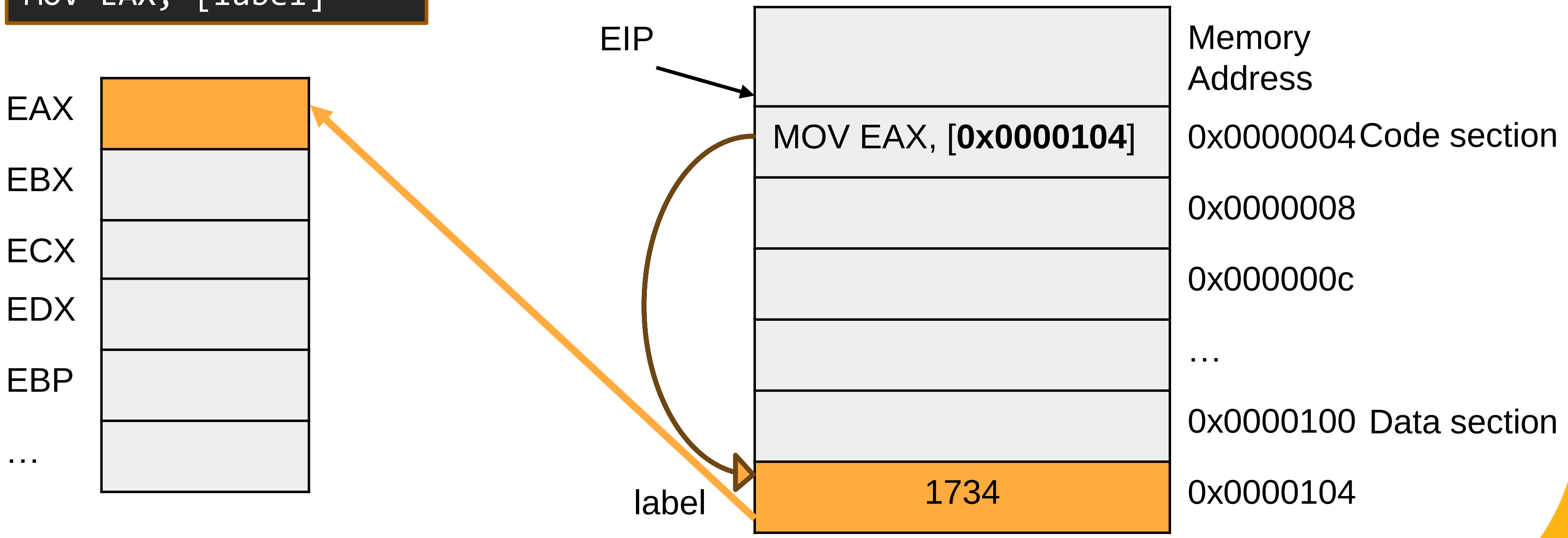
```
MOV EAX, [ECX]
```

Note: Brackets are used to dereference: the retrieve the value at this address



Memory to register

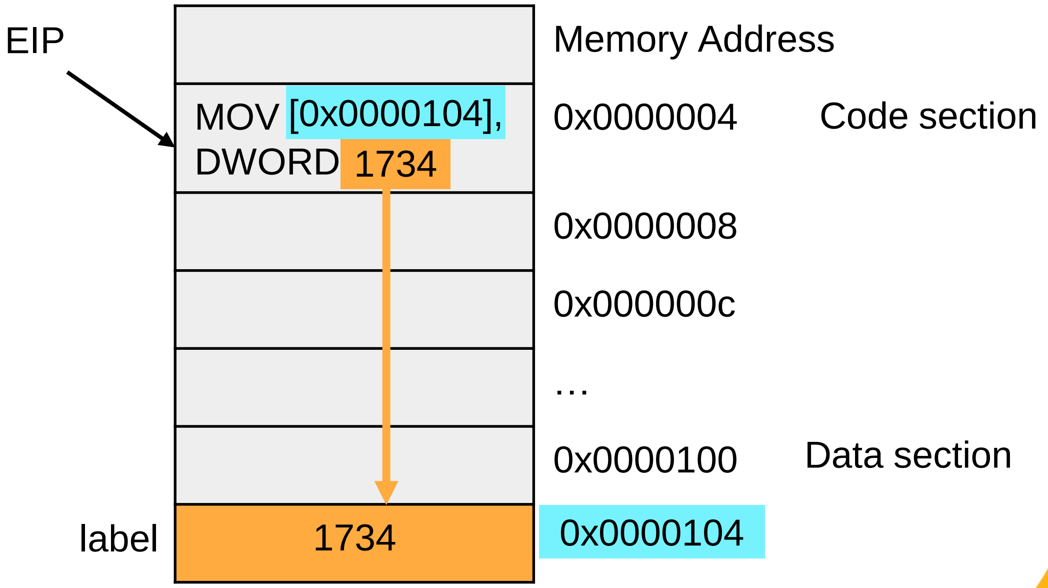
```
MOV EAX, [0x0000104]
```
or
```
MOV EAX, [label]
```



Immediate to memory

```
MOV [0x0000104], DWORD 1734
```

```
MOV [label], DWORD 1734
```

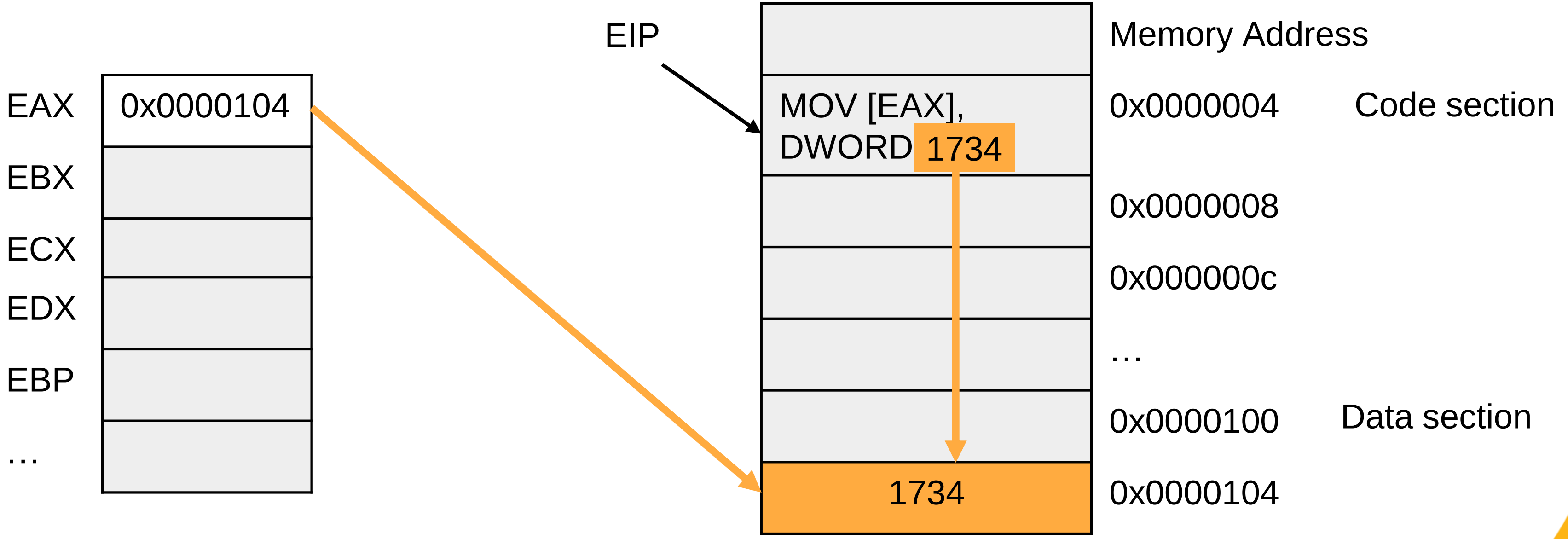


Addressing keywords
BYTE: 1 byte
WORD: 2 bytes
DWORD: 4 bytes
QWORD: 8 bytes
DWORD_PTR: represents an address

- 32 bits on 32 bit system (4 bytes)
- 64 bits on 64 bit system (8 bytes)

Immediate to register indirect

```
MOV [EAX], DWORD 1734
```



Summary of addressing formats

```
MOV EAX, 1734                 ; EAX = 1734

MOV EAX, ECX                  ; EAX = ECX

MOV EAX, [0x0000104]          ; EAX = value at 0x0000104

MOV EAX, [label]              ; EAX = value at address pointed to by label

MOV EAX, [ECX]                ; EAX = value at address stored in ECX

MOV [0x0000104], DWORD 1734   ; move value 1734 to address 0x0000104

MOV [label], DWORD 1734       ; move value 1734 to address pointed to by label

MOV [EAX], DWORD 1734         ; move value 1734 to address stored in EAX
```

Example

```
section .data
        x dd 13
        y dd 0

section .text
main:   mov eax, [x]
        mov ebx, x
        inc eax
        mov [y], eax
global main
```

- What will this do?

Example

```
section .data
        x dd 13
        y dd 0

section .text
main:   mov eax, [x] ; moves value of x into eax. Eax = 13
        mov ebx, x   ; moves address of x into ebx.
        inc eax      ; eax = 13+1 = 14
        mov [y], eax ; moves value of eax into memory location that y points to.
global main          ; indicates beginning of program

; This program does: y = x + 1
```

Addressing formats not allowed

```
Add [mem1] [mem2]
Mov [mem1] [mem2]
```

- Memory to memory addressing is not allowed in x86
- You need to use an immediate value or a register as the step in between moving memory to memory

Clearing bits

- How do I clear the lower 4 bits of the AL register (8 bits long)?

- Answer: use AND instruction – Logical “and” two numbers
  ANDing by 0 is always zero, ANDing by 1 preserves the other number

```
AND AL, 0xF0
```

```
    1010 1010 = what was in AL before
AND 1111 0000 = 0xF0
    1010 0000 = result, lower 4 bits cleared
```

Setting bits

- How do I set the lower 4 bits of the AL register?

- Answer: use OR instruction – Logical “or” two numbers
  ORing by 1 is always one, ORing by 0 preserves the other number

```
OR AL, 0x0F
```

```
    1010 1010 = what was in AL before
 OR 0000 1111 = 0x0F
    1010 1111 = result, lower 4 bits set to 1
```

Shift instructions

- SHL – Logical Shift left
- SHR – Logical Shift right
- SAL – Arithmetic shift left
- SAR – Arithmetic shift right

SHL/SAL

- Logical shift left and arithmetic shift left are the same operation
- The carry flag gets set to the value being shifted out
- A zero get carried into the new vacancy on the right

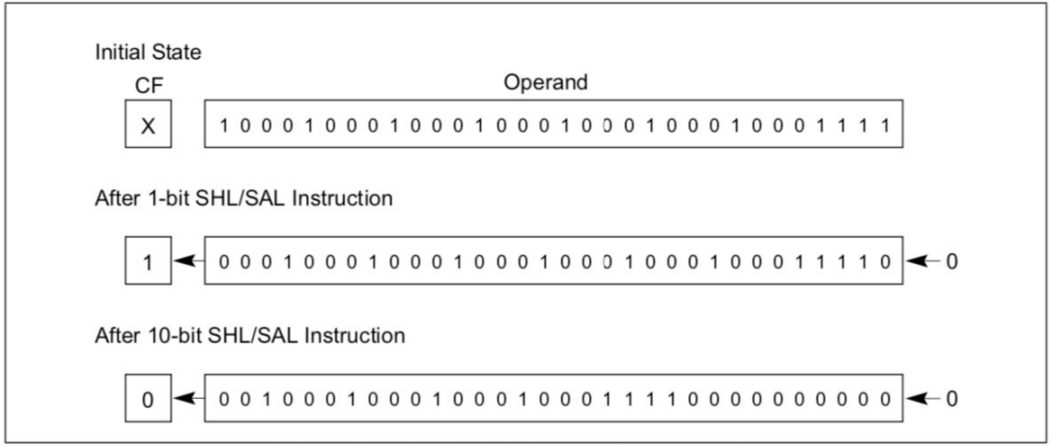


Figure 7-6. SHL/SAL Instruction Operation

SHR vs SAR

- Logical shift right (SHR) and Arithmetic shift right (SAR) are not the same.
- SHR and SAR both shift the LSB into the Carry flag during shifting
- SHR always carries in a zero to the MSB.
- SAR carries a 0 into MSB for positive numbers and a 1 for negative numbers.

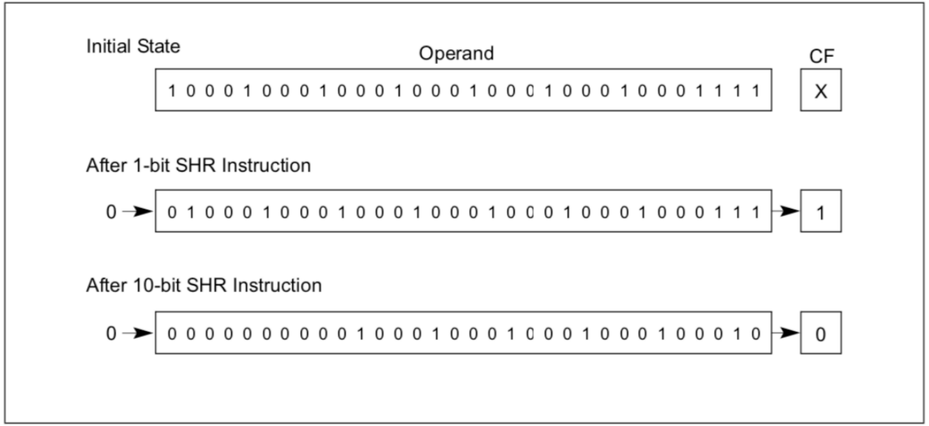


Figure 7-7. SHR Instruction Operation

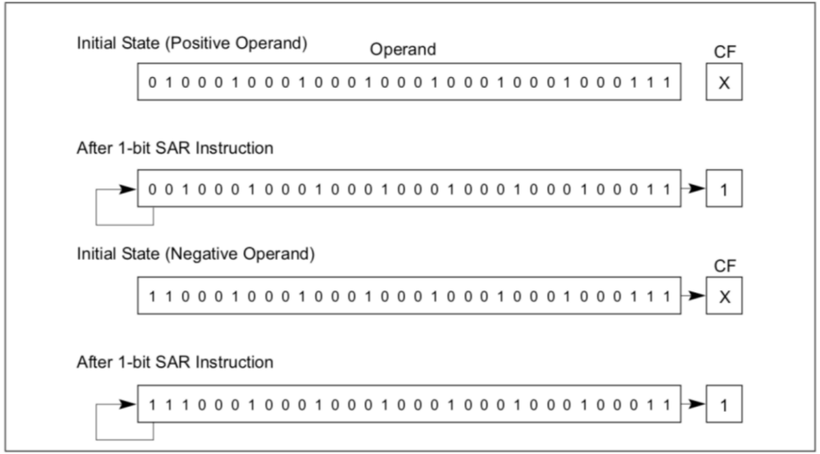


Figure 7-8. SAR Instruction Operation

Rotate instructions

- ROL – logical Rotate left
- ROR – logical Rotate right
- RCL – rotate through carry left
- RCR – rotate through carry right

ROL vs RCL and ROR vs RCR

- ROR and ROL do not include CF as an element in the rotation, but they do copy the bit that rotates over into CF
- RCR and RCL include the CF as one of the elements in the rotation. The CF flag gets set automatically along the way.

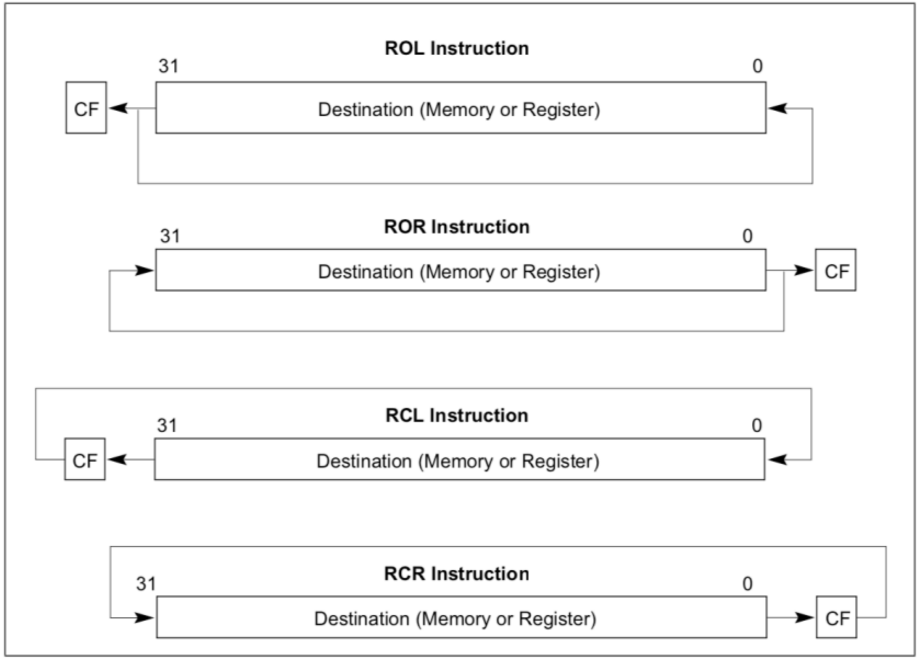


Figure 7-10. ROL, ROR, RCL, and RCR Instruction Operations

Control Instructions

- JMP – jump
- JE/JZ – jump if equal/jump if zero
- JNE/JNZ – jump if not equal/jump if not zero
- JG/JNLE – jump if greater than/jump if not less than or equal
- JGE/JNL – jump if greater than or equal/jump if not less than
- CALL – call a procedure
- RET – return

String instructions

These operate on chunks of data, not really for “strings” in the traditional sense.

These are sometimes harder to use, since you need to keep track of information in chunks

- MOVS/MOVSB/MOVSW/MOVSD – move string
- CMPS/CMPSB/CMPSW/CMPSD – compare string
- LODS/LODSB/LODSW/LODSD – load string
- STOS/STOSB/STOSW/STOSD – store string

NASM pseudo instructions

- Different from intel x86 arch instructions
- They are still written in the same .asm file. The assembler will correctly interpret which keywords are NASM pseudo instructions and which are x86
- .data section uses keywords for integers: DB, DW, DD, DQ
- .data section uses keywords for floats: DT, DTQ

```
; testdata_64.asm  a program to demonstrate data types and values
; assemble:     nasm -f elf64 -l testdata_64.lst  testdata_64.asm
; link:         gcc -m64 -o testdata_64  testdata_64.o
; run:          ./testdata_64
; Look at the list file, testdata_64.lst
; no output
; Note! nasm ignores the type of data and type of reserved
; space when used as memory addresses.
; You may have to use qualifiers BYTE, WORD, DWORD or QWORD

        section .data           ; data section
                                ; initialized, writeable

                                ; db for data byte, 8-bit
db01:   db      255,1,17        ; decimal values for bytes
db02:   db      0xff,0ABh       ; hexadecimal values for bytes
db03:   db      'a','b','c'     ; character values for bytes
db04:   db      "abc"           ; string value as bytes 'a','b','c'
db05:   db      'abc'           ; same as "abc" three bytes
db06:   db      "hello",13,10,0 ; "C" string including cr and lf

                                ; dw for data word, 16-bit
dw01:   dw      12345,-17,32    ; decimal values for words
dw02:   dw      0xFFFF,0abcdH   ; hexadecimal values for words
dw03:   dw      'a','ab','abc'  ; character values for words
dw04:   dw      "hello"         ; three words, 6-bytes allocated

                                ; dd for data double word, 32-bit
dd01:   dd      123456789,-7    ; decimal values for double words
dd02:   dd      0xFFFFFFFF      ; hexadecimal value for double words
dd03:   dd      'a'             ; character value in double word
dd04:   dd      "hello"         ; string in two double words
dd05:   dd      13.27E30        ; floating point value 32-bit IEEE

                                ; dq for data quad word, 64-bit
dq01:   dq      123456789012,-7 ; decimal values for quad words
dq02:   dq      0xFFFFFFFFFFFFFFFF ; hexadecimal value for quad words
dq03:   dq      'a'             ; character value in quad word
dq04:   dq      "hello_world"   ; string in two quad words
dq05:   dq      13.27E300       ; floating point value 64-bit IEEE

                                ; dt for data ten of 80-bit floating point
dt01:   dt      13.270E3000     ; floating point value 80-bit in register
```

NASM pseudo instructions (cont.)

- .bss section uses keywords: resb, resw, resd, resq

```
        section .bss            ; reserve storage space
                                ; uninitialized, writeable

s01:    resb    10              ; 10 8-bit bytes reserved
s02:    resw    20              ; 20 16-bit words reserved
s03:    resd    30              ; 30 32-bit double words reserved
s04:    resq    40              ; 40 64-bit quad words reserved
s05:    resb    1               ; one more byte
```

- equ is used to assign a constant value to a symbol. These are different from labels since labels are an address and can point to changing data.

```
section .data
    ; Define a constant for the value of pi
    pi1 dd 3.14159 ; 32 bit float
    pi2 equ 3.14159

section .text
    global main

main:
    ; Load the value of pi into a register
    mov EAX, dword [pi1] ; move 3.14159 into EAX
    mov EBX, dword pi2 ; move 3.14159 into EAX
```

Endianness

- The order that data is stored in memory
- **Little Endian**: least significant byte is stored at earlier address
  - Intel is stored in little endian
- **Big Endian**: the most significant byte is stored at earlier address

Data

`0x12345678`

Stored as

Little Endian

`0x 78 56 34 12`

Big Endian

`0x 12 34 56 78`

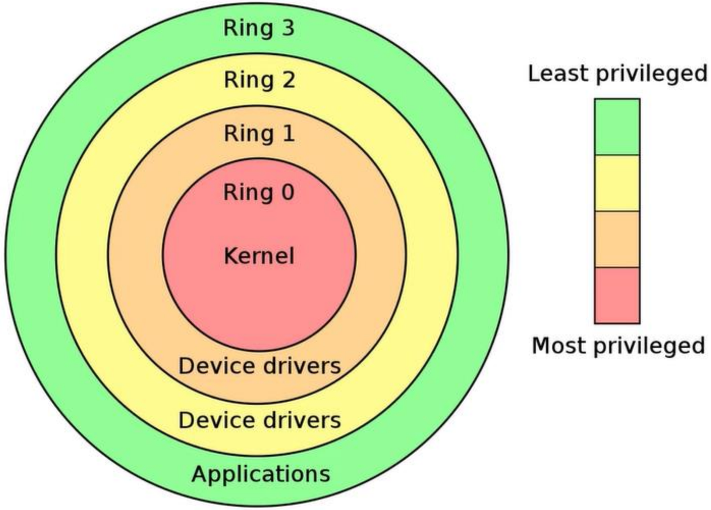
In memory

Ascii

- The terminal prints out characters in ascii.
- Characters are also read in in ascii. That means if you are entering a number, the value will be stored as ascii and might need to be converted to an actual number
- To convert ascii → number, subtract 48
- For multi-digit numbers, you must consider how you will reconstruct the whole number. Therefore, each digit may need to individually be converted to be interpreted.
- Ascii uses 1 byte to represent each character

Privilege mode

- Difference between kernel mode and user mode is the privilege
- There are 4 privilege levels on x86
- CPL register stores the current privilege level. Not general purpose.
- Privileged instructions can only execute when CPL is 0.
- Kernel is the only one that can grant access to memory.



System call

- A **system call** is way of requesting the kernel to do something for the user because the user doesn’t have privileges for everything.
- A system call is basically a function with parameters

- Example:

```
ssize_t write(int fd, const void *buf, size_t count);
```

- ssize_t is a type defined by the OS in types.h used for the return value
- **write** is the name of the system call and can be found in the file unistd_64.h
- The rest are parameters according to what the system call does and needs

System call (cont.)

- To use System calls in intel 64 bit assembly you need to:
- Put the system call number in the RAX register
- The rest of the parameters are put in the following registers in order from left to right: rdi, rsi, rdx, r10, r8, r9
- Extra parameters are placed on the stack
- Once set up, use the **syscall** instruction to call the system call.

SYSCALL SETUP	
Location	**Syscall reg purpose**
RAX	Syscall number
RDI	1st argument
RSI	2nd argument
RDX	3rd argument
R10	4th argument
R8	5th argument
R9	6th argument
Stack	7th + arguments

Don’t ask me why R8 comes before R9, I didn’t write 64-bit x86

Syscall "write" example

- If I want to print to the screen, I have to use the **write** syscall.
- Read the docs for **write**: https://manpages.debian.org/unstable/manpages-dev/write.2.en.html

```
ssize_t write(int fd, const void *buf, size_t count);
```

- Write has 3 parameters:
  - fd – the file descriptor
  - buf – the string to write
  - count – the length of the string
- The write syscall is assigned the number 1
- Therefore, the register setup will look like this:

Reg	Value	explanation
RAX	1	Syscall number
RDI	1	Std-out file descriptor
RSI	Address to string	String to print
RDX	Length of string	Number of chars to print

File descriptors

- A **file descriptor** is a handle that an operating system uses to access files, sockets, or other input/output (I/O) resources.
- In Linux, file descriptors are represented as non-negative integers.
- There are 3 standard streams that are typically pre-opened by the operating system when a program starts, and they serve as default channels for input and output.

fd#	Name	Purpose
0	Standard in (std-in)	Take input from terminal
1	Standard out (std-out)	Write output to terminal
2	Standard error (std-err)	Catalog errors to the terminal

- Syscalls typically use file descriptors to identify which resources are being used

Common system calls

Syscall name	Number	Description
Read	0	Read from file descriptor
Write	1	Write to file descriptor
Open	2	Open a file
Close	3	Close a file
Exit	60	Exit a program with an exit code

- You'll always need the exit syscall so your program doesn't seg fault. Seg faults happens because the program continues to read the next instruction after the end of the program
- More syscalls here: https://filippo.io/linux-syscall-table/
- You can find the syscall list on GL in file **/usr/include/asm/unistd_64.h**

Hello world example

```
section .data
    msg db "Hello World!", 10, 0

section .text
    global main

main:
    ; Print "Hello, World!" message
    mov     rax, 1                  ; Syscall number for sys_write
    mov     rdi, 1                  ; File descriptor 1 (stdout)
    mov     rsi, msg                ; Load address of the message (not value)
    mov     rdx, 13                 ; Length of the message
    syscall                         ; Invoke syscall to write the message

    ; Exit the program
    mov     rax, 60                 ; Syscall number for sys_exit
    xor     rdi, rdi                ; Exit code 0
    syscall                         ; Invoke syscall to exit
```

Hello world example with computed string length

```
section .data
    msg db "Hello World!", 10, 0
    msg_len equ $ - msg

section .text
    global main

main:
    ; Print "Hello, World!" message
    mov     rax, 1              ; Syscall number for sys_write
    mov     rdi, 1              ; File descriptor 1 (stdout)
    mov     rsi, msg            ; Load address of the message (not value)
    mov     rdx, msg_len        ; Length of the message
    syscall                     ; Invoke syscall to write the message

    ; Exit the program
    mov     rax, 60             ; Syscall number for sys_exit
    xor     rdi, rdi            ; Exit code 0
    syscall                     ; Invoke syscall to exit
```

Note: **equ** computes a value and does not store the value at any address. At assemble-time, the value is substituted into the locations in code where it is needed.

Note: **$** computes the address where the $ is located. $ - msg computes the difference between the current address and the address pointed to by msg

Syscall on 32-bit

- System calls will be different on different OS
- So is the location of the unistd.h file
- On intel the values of the calls are different between 32 and 64 bit
- The registers that take the arguments are also different between 64 and 32 bit
- 32 bit argument registers are:
  - EAX gets the call number
  - EBX gets the first argument
  - ECX gets the second
  - EDX gets the third
  - EDI gets the fourth
  - ESI gets the fifth
- The system call is made by running the command int 80h
- May still work on 64 bit architecture

Global main vs global _start

- You may see _start online as the global entry point to an asm program

- When you specify _**start** as the entry point, you're essentially bypassing the C runtime startup code provided by the compiler/linker.
- With _start, you're responsible for setting up the environment for your program, such as initializing registers, setting up the stack

- When you specify **main** as the entry point, you're relying on the C runtime startup code provided by the compiler/linker.
- The C runtime startup code handles various initialization tasks such as setting up the environment, initializing global variables, parsing command-line arguments (if any), and eventually calling your main function.

Words of wisdom

References

- Ivan Sekyonda’s slides