MONTANA STATE UNIVERSITY

8/28/2025

Malicious Code Analysis

Fangtian Zhong
CSCI 591

Gianforte School of Computing
Norm Asbjornson College of Engineering
E-mail: fangtian.zhong@montana.edu

Overview

01 Arithmetic Instructions

02 Logical Instructions

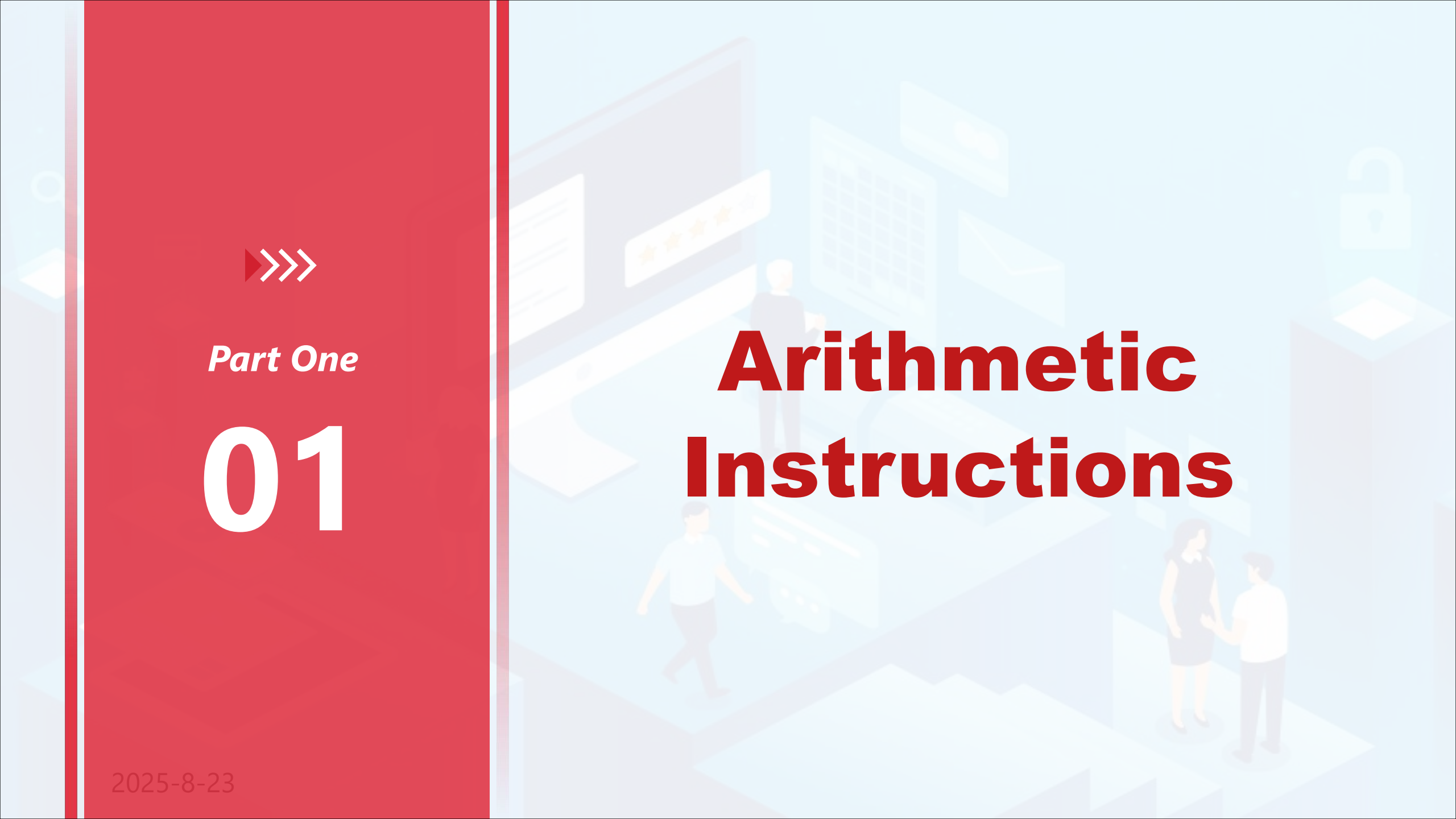
03 Conditions

04 Loops

Part One

01

Arithmetic Instructions



2025-8-23

The INC Instruction

- The INC instruction is used for incrementing an operand by one. It works on a single operand that can be either in a register or in memory.
- The INC instruction has the following syntax −
  - INC destination
- The operand destination could be an 8-bit, 16-bit, 32-bit, or 64-bit operand.
- ***Example***

```
INC RBX;       Increments 64-bit register
INC DL;        Increments 8-bit register
INC [count] ;  Increments the count variable
```

Example: INC

```
bits 64
default rel

section .data
    message db "Test INC!",0xd, 0xa, 0
    format db "%llu", 0xd, 0xa, 0; printing format for qword numbers

section .bss
    uninitializedData resq 1

section .text
    extern printf
    global main
    extern ExitProcess
    extern _CRT_INIT
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
```

```
;Print the string
lea rcx, [message]
call printf

; Print the constant value
mov qword [uninitializedData], 42
INC qword [uninitializedData]
mov rcx, format
mov dl, byte [uninitializedData]
call printf


; Exit the program
xor rax, rax
call ExitProcess
```

The DEC Instruction

- The DEC instruction is used for decrementing an operand by one. It works on a single operand that can be either in a register or in memory.
- The DEC instruction has the following syntax −
  - **DEC destination**
- The operand destination could be an 8-bit, 16-bit, 32-bit or 64-bit operand.

Example: DEC

```
bits 64
default rel

section .data
    message db "Test DEC!",0xd, 0xa, 0
    format db "%llu", 0xd, 0xa, 0; printing format for qword numbers

section .bss
    uninitializedData resq 1

section .text
    extern printf
    global main
    extern ExitProcess
    extern _CRT_INIT
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
```

```
    ;Print the string
    lea rcx, [message]
    call printf

    ; Print the constant value
    mov qword [uninitializedData], 42
    INC qword [uninitializedData]
    DEC qword [uninitializedData]
    mov rcx, format
    mov dl, byte [uninitializedData]
    call printf

    ; Exit the program
    xor rax, rax
    call ExitProcess
```

The ADD and SUB Instructions

- The ADD and SUB instructions are used for performing simple addition/subtraction of binary data in byte, word and doubleword size, i.e., for adding or subtracting 8-bit, 16-bit, 32-bit or 64-bit operands, respectively.
- The ADD and SUB instructions have the following syntax −
  - ADD/SUB destination, source
- The ADD/SUB instruction can take place between −
  - Register to register
  - Memory to register
  - Register to memory
  - Register to constant data
  - Memory to constant data

Examples

```
bits 64
default rel

section .data
    resultMsg db "The result is: %lld", 0

section .text
    global main

extern printf
extern ExitProcess
extern _CRT_INIT
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT

    ; Perform the signed subtraction
    mov rax, 5
    mov rbx, 10
    sub rax, rbx
```

```
    ; Print the result
    mov rcx, resultMsg
    mov rdx, rax
    call printf

    mov rax, 5
    mov rbx, 10
    add rax, rbx
    ; Print the result
    mov rcx, resultMsg
    mov rdx, rax
    call printf

    ; Exit the program
    xor rax, rax
    call ExitProcess
```

The MUL/IMUL Instruction

- There are two instructions for multiplying binary data. The MUL (Multiply) instruction handles unsigned data and the IMUL (Integer Multiply) handles signed data. Both instructions affect the Carry and Overflow flag.
- The syntax for the MUL/IMUL instructions is as follows −
  - MUL/IMUL multiplier
- Multiplicand in both cases will be in an accumulator, depending upon the size of the multiplicand and the multiplier and the generated product is also stored in two registers depending upon the size of the operands.

Different Cases

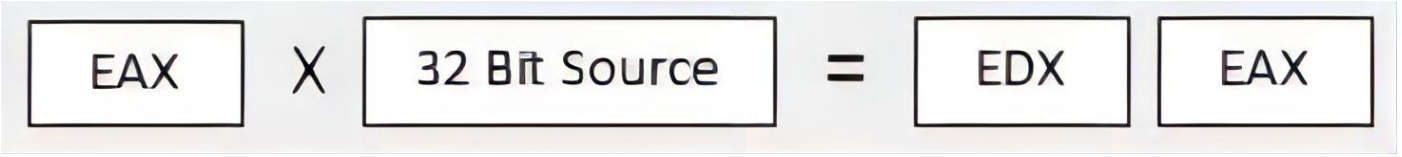
- When two bytes are multiplied −
- The multiplicand is in the AL register, and the multiplier is a byte in the memory or in another register. The product is in AX. High-order 8 bits of the product is stored in AH and the low-order 8 bits are stored in AL.

Different Cases

- When two one-word values are multiplied −
- The multiplicand should be in the AX register, and the multiplier is a word in memory or another register. For example, for an instruction like MUL DX, you must store the multiplier in DX and the multiplicand in AX.
- The resultant product is a doubleword, which will need two registers. The high-order (leftmost) portion gets stored in DX and the lower-order (rightmost) portion gets stored in AX.

Different Cases

- When two doubleword values are multiplied, the multiplicand should be in EAX and the multiplier is a doubleword value stored in memory or in another register. The product generated is stored in the EDX:EAX registers, i.e., the high order 32 bits gets stored in the EDX register and the low order 32-bits are stored in the EAX register.



Different Cases

- When two qword values are multiplied, the multiplicand should be in RAX and the multiplier is a qword value stored in memory or in another register. The product generated is stored in the RDX:RAX registers, i.e., the high order 64 bits gets stored in the RDX register and the low order 64 bits are stored in the RAX register.

Examples

- MOV AL, 10
- MOV DL, 25
- MUL DL
- ...
- MOV DL, 0FFH ; DL= -1
- MOV AL, 0BEH ; AL = -66
- IMUL DL

Examples

```
bits 64
default rel

section .data
    resultMsg db "The result is: %lld", 0

section .text
    global main

extern printf
extern ExitProcess
extern _CRT_INIT
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT

    ; Perform unsigned multiplication
    mov rax, 5
    mov rbx, 7
    mul rbx
```

```
    ; Print the result
    mov rcx, resultMsg
    mov rdx, rax
    call printf

    ; Perform signed multiplication
    mov rax, 5
    mov rbx, -7
    imul rax, rbx

    ; Print the result
    mov rcx, resultMsg
    mov rdx, rax
    call printf

    ; Exit the program
    xor rax, rax
    call ExitProcess
```

The DIV/IDIV Instructions

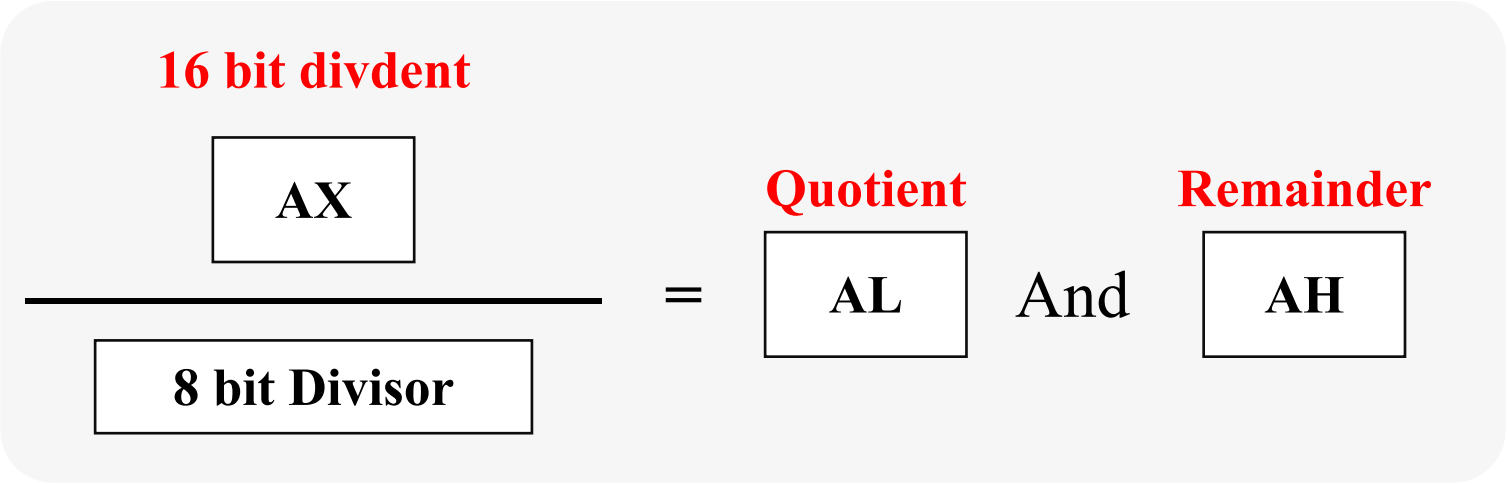
- The DIV (Divide) instruction is used for unsigned data and the IDIV (Integer Divide) is used for signed data.
- The format for the DIV/IDIV instruction −
  - DIV/IDIV divisor

The DIV/IDIV Instructions

- The dividend is in an accumulator. Both the instructions can work with 8-bit, 16-bit, 32-bit, or 64 bit operands. The operation affects all six status flags. Following section explains four cases of division with different operand size −

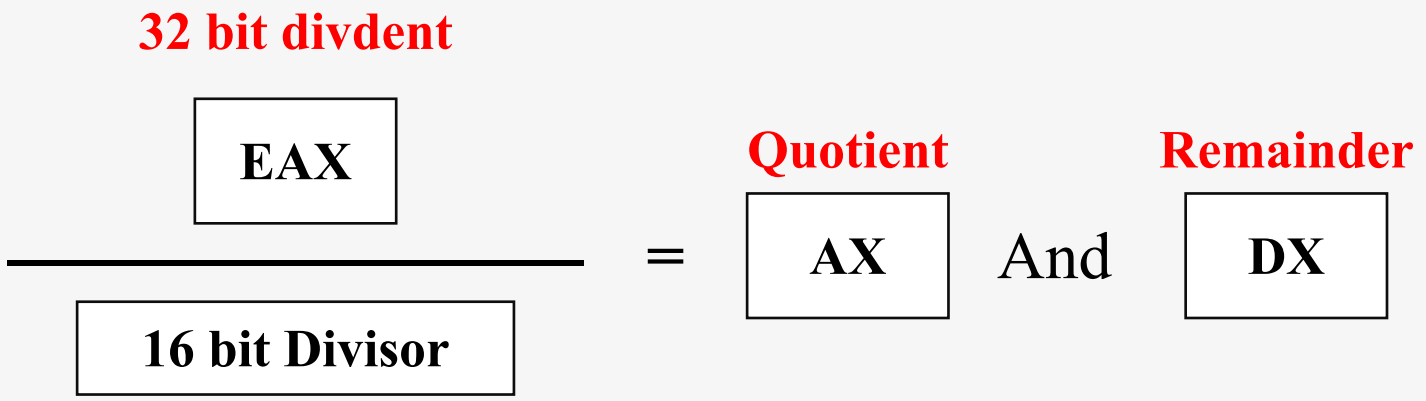
Different Cases

- When the divisor is 1 byte −
- The dividend is assumed to be in the AX register (16 bits). After division, the quotient goes to the AL register and the remainder goes to the AH register.



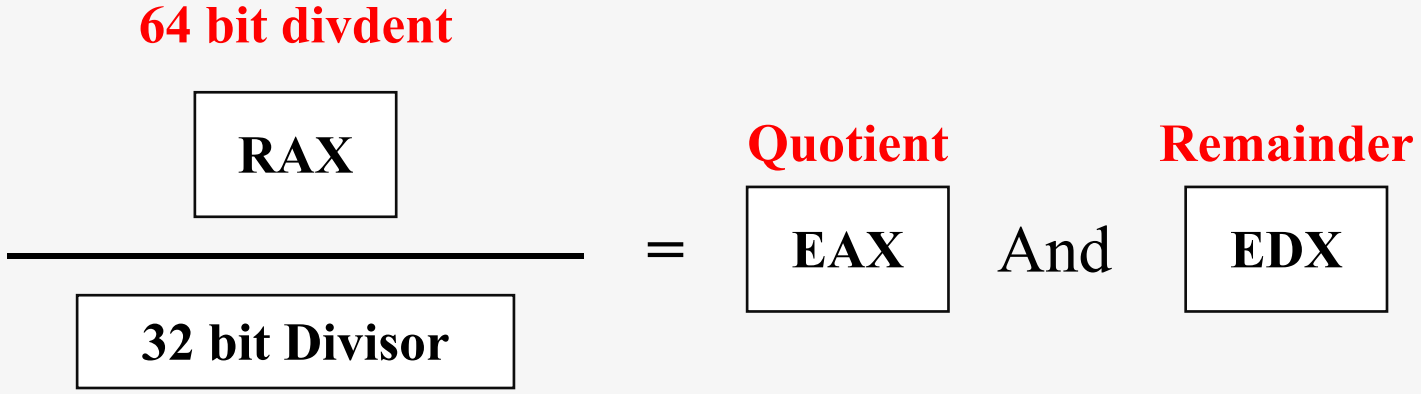
Different Cases

- When the divisor is 1 word −
- The dividend is assumed to be 32 bits long and in the EAX registers. After division, the 16-bit quotient goes to the AX register and the 16-bit remainder goes to the DX register.



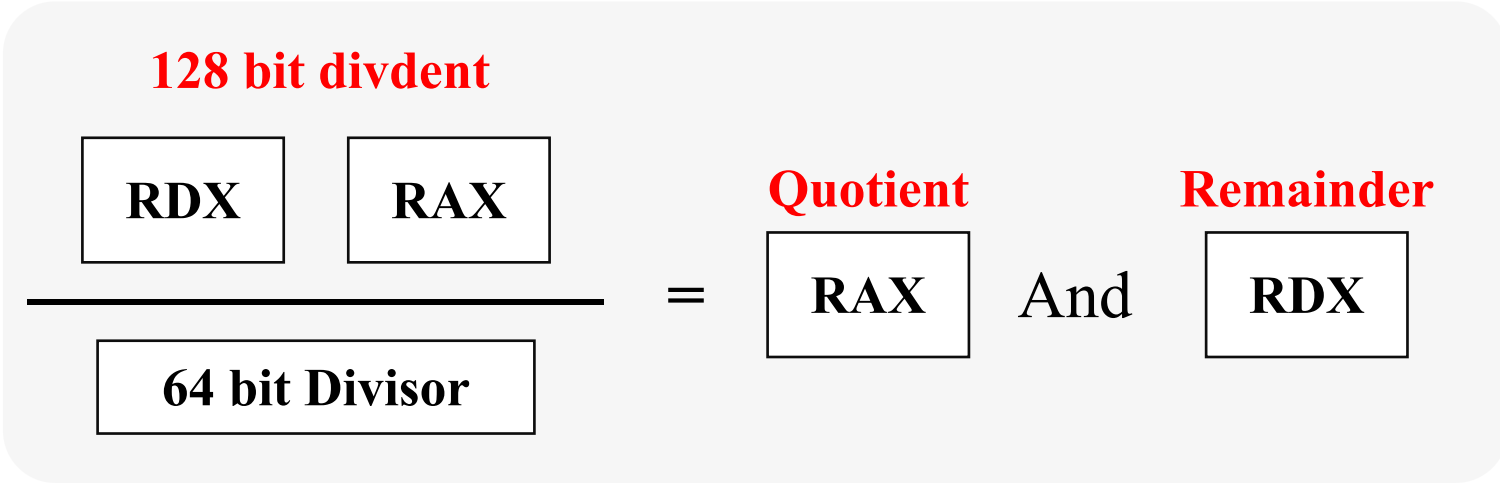
Different Cases

- When the divisor is doubleword −
- The dividend is assumed to be 64 bits long and in the RAX registers. After division, the 32-bit quotient goes to the EAX register and the 32-bit remainder goes to the EDX register.



Different Cases

- When the divisor is qword −
- The dividend is assumed to be 128 bits long and in the RAX registers. The high-order 64 bits are in RDX and the low-order 64 bits are in RAX. After division, the 64-bit quotient goes to the RAX register and the 64-bit remainder goes to the RDX register.



Examples

```
bits 64
default rel

section .data
    quotientMsg db "The quotient is: %lld", 0xd, 0xa, 0
    remainderMsg db "The remainder is: %lld", 0xd, 0xa, 0

section .text
global main
extern printf
extern ExitProcess
extern _CRT_INIT

main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT

    ; Perform unsigned division
    mov rax, 15
    mov rbx, 3
    xor rdx, rdx  ; Clear RDX to zero for the remainder
    div rbx
```

```
    ; The quotient is stored in RAX and the remainder in RDX

    ; Print the quotient
    mov rcx, quotientMsg
    mov rdx, rax
    call printf

    ; Print the remainder
    mov rcx, remainderMsg
    mov rdx, rdx
    call printf


    ;Exit the program
    xor rax, rax
    call ExitProcess
```

Part Two

02

Logical Instructions

2025-8-23

Logical Instructions

- The processor instruction set provides the instructions AND, OR, XOR, TEST, and NOT Boolean logic, which tests, sets, and clears the bits according to the need of the program.
- The format for these instructions −

Sr.No.	Instruction	Format
1	AND	AND operand1, operand2
2	OR	OR operand1, operand2
3	XOR	XOR operand1, operand2
4	TEST	TEST operand1, operand2
5	NOT	NOT operand1

- The first operand in all the cases could be either in register or in memory. The second operand could be either in register/memory or an immediate (constant) value. However, memory-to-memory operations are not possible.
- These instructions compare or match bits of the operands and set the CF, OF, PF, SF and ZF flags.

The AND Instruction

- The AND instruction is used for supporting logical expressions by performing bitwise AND operation. The bitwise AND operation returns 1, if the matching bits from both the operands are 1, otherwise it returns 0. For example −

```
                Operand1:  0101
                Operand2:  0011
----------------------------
After AND -> Operand1:  0001
```

- The AND operation can be used for clearing one or more bits. For example, say the BL register contains 0011 1010. If you need to clear the high-order bits to zero, you AND it with 0FH.
- AND BL, 0FH ; This sets BL to 0000 1010

The AND Instruction

- Let's take up another example. If you want to check whether a given number is odd or even, a simple test would be to check the least significant bit of the number. If this is 1, the number is odd, else the number is even.
- Assuming the number is in AL register, we can write −

```
AND     AL, 01H      ; ANDing with 0000 0001
JZ    EVEN_NUMBER
```

Examples

```
default rel

section .data
    num1 dd 0x2
    num2 dd 0x1
    format db "%x", 0
    evenNum db "this is a even number!", 0xd, 0xa, 0

section .bss
    result resd 1

section .text
    extern printf
    global main
    extern _CRT_INIT
    extern ExitProcess
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
```

```
    ; Perform bitwise AND between num1 and num2
    mov eax, [num1]    ; Load num1 into EAX
    and eax, [num2]    ; Perform AND operation with num2 and store the result in EAX
    jz evnn

    mov [result], eax  ; Store the result in result variable
    ; Print the result in binary format
    mov rcx, format
    mov rdx, [result]
    call printf
    jmp outprog

evnn:
    mov rcx, evenNum
    call printf

outprog:
    ; Exit the program
    xor eax, eax
    call ExitProcess
```

The TEST Instruction

- The TEST instruction works same as the AND operation, but unlike AND instruction, it does not change the first operand. So, if we need to check whether a number in a register is even or odd, we can also do this using the TEST instruction without changing the original number.

```
TEST    AL, 01H
JZ      EVEN_NUMBER
```

The NOT Instruction

- The NOT instruction implements the bitwise NOT operation. NOT operation reverses the bits in an operand. The operand could be either in a register or in the memory.

- **For example,**

```
                Operand1:     0101 0011
After NOT -> Operand1:     1010 1100
```

Part Three

03

Conditions

Condition Execution

- Conditional execution in assembly language is accomplished by several looping and branching instructions. These instructions can change the flow of control in a program. Conditional execution is observed in two scenarios −
  - **Unconditional jump**
    - This is performed by the JMP instruction. Conditional execution often involves a transfer of control to the address of an instruction that does not follow the currently executing instruction. Transfer of control may be forward, to execute a new set of instructions or backward, to re-execute the same steps.
  - **Conditional jump**
    - This is performed by a set of jump instructions j<condition> depending upon the condition. The conditional instructions transfer the control by breaking the sequential flow and they do it by changing the offset value in IP.

CMP Instruction

- The CMP instruction compares two operands. It is generally used in conditional execution. This instruction basically subtracts one operand from the other for comparing whether the operands are equal or not. It does not disturb the destination or source operands. It is used along with the conditional jump instruction for decision making.
- Syntax
  - CMP destination, source
- CMP compares two numeric data fields. The destination operand could be either in register or in memory. The source operand could be a constant (immediate) data, register or memory.
- **Example**

```
CMP RAX,        00  ; Compare the DX value with zero
JE  L7      ; If yes, then jump to label L7
.
.
L7: ...
```

CMP Instruction

- CMP is often used for comparing whether a counter value has reached the number of times a loop needs to be run.
- Consider the following typical condition −
- **Example**

```
INC    RDX

CMP    RDX, 10        ; Compares whether the counter has reached 10

JLE    LP1     ; If it is less than or equal to 10, then jump to LP1
```

Unconditional Jump

- The JMP instruction provides a label name where the flow of control is transferred immediately. The syntax of the JMP instruction is −

```
JMP  label
```

Examples

```
default rel

section .data
    num1 dd 0x2
    num2 dd 0x1
    format db "%x", 0
    evenNum db "this is a even number!", 0xd, 0xa, 0

section .bss
    result resd 1

section .text
    extern printf
    global main
    extern _CRT_INIT
    extern ExitProcess
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
```

```
    ; Perform bitwise AND between num1 and num2
    mov eax, [num1]    ; Load num1 into EAX
    and eax, [num2]    ; Perform AND operation with num2 and store the result in EAX
    jz evnn

    mov [result], eax  ; Store the result in result variable
    ; Print the result in binary format
    mov rcx, format
    mov rdx, [result]
    call printf
    jmp outprog

evnn:
    mov rcx, evenNum
    call printf

outprog:
    ; Exit the program
    xor eax, eax
    call ExitProcess
```

Conditional Jump

- If some specified condition is satisfied in conditional jump, the control flow is transferred to a target instruction. There are numerous conditional jump instructions depending upon the condition and data.
- Following are the conditional jump instructions used on signed data used for arithmetic operations −

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JG/JNLE	Jump Greater or Jump Not Less/Equal	OF, SF, ZF
JGE/JNL	Jump Greater/Equal or Jump Not Less	OF, SF
JL/JNGE	Jump Less or Jump Not Greater/Equal	OF, SF
JLE/JNG	Jump Less/Equal or Jump Not Greater	OF, SF, ZF

Conditional Jump

- Following are the conditional jump instructions used on unsigned data used for logical operations −

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JA/JNBE	Jump Above or Jump Not Below/Equal	CF, ZF
JAE/JNB	Jump Above/Equal or Jump Not Below	CF
JB/JNAE	Jump Below or Jump Not Above/Equal	CF
JBE/JNA	Jump Below/Equal or Jump Not Above	AF, CF

Conditional Jump

- The following conditional jump instructions have special uses and check the value of flags −

Instruction	Description	Flags tested
JXCZ	Jump if CX is Zero	none
JC	Jump If Carry	CF
JNC	Jump If No Carry	CF
JO	Jump If Overflow	OF
JNO	Jump If No Overflow	OF
JP/JPE	Jump Parity or Jump Parity Even	PF
JNP/JPO	Jump No Parity or Jump Parity Odd	PF
JS	Jump Sign (negative value)	SF
JNS	Jump No Sign (positive value)	SF

Syntax

★ The syntax for the J<condition> set of instructions −

★ **Example,**

```
CMP     AL, BL
JE      EQUAL
CMP     AL, BH
JE      EQUAL
CMP     AL, CL
JE      EQUAL
NON_EQUAL: ...
EQUAL: ...
```

Part Four

04

Loops

Loops

The JMP instruction can be used for implementing loops. For example, the following code snippet can be used for executing the loop-body 10 times.

```
MOV     CX, 10
L1:
<LOOP-BODY>
DEC     CX
JNZ     L1
```

Loops

- The processor instruction set, however, includes a group of loop instructions for implementing iteration.
- The basic LOOP instruction has the following syntax −
  - LOOP label
- Where, label is the target label that identifies the target instruction as in the jump instructions. The LOOP instruction assumes that the ECX register contains the loop count. When the loop instruction is executed, the ECX register is decremented and the control jumps to the target label, until the ECX register value, i.e., the counter reaches the value zero.
- The above code snippet could be written as −

```
mov ECX,10
l1:
<loop body>
loop l1
```

Examples

```
bits 64
default rel

section .data
    format db "%d", 0
    newline db 0x0A, 0x00
section .bss
    temrcx resq 1

section .text
    extern printf
    global main
    extern _CRT_INIT
    extern ExitProcess
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT

    ; Initialize loop counter
    mov rcx, 10
```

```
loop_start:
    ; Print the current number
    mov qword [temrcx], rcx
    mov rcx, format
    mov rdx, qword [temrcx]
    call printf


    ; Print newline character
    lea rcx, newline
    call printf

    mov rcx, qword [temrcx]
    ; Decrement loop counter and check if it's zero
    loop loop_start

    ; Exit the program
    xor eax, eax
    call ExitProcess
```

THE END

Fangtian Zhong

CSCI 591

Gianforte School of Computing
Norm Asbjornson College of Engineering
E-mail: fangtian.zhong@montana.edu

8/28/2025