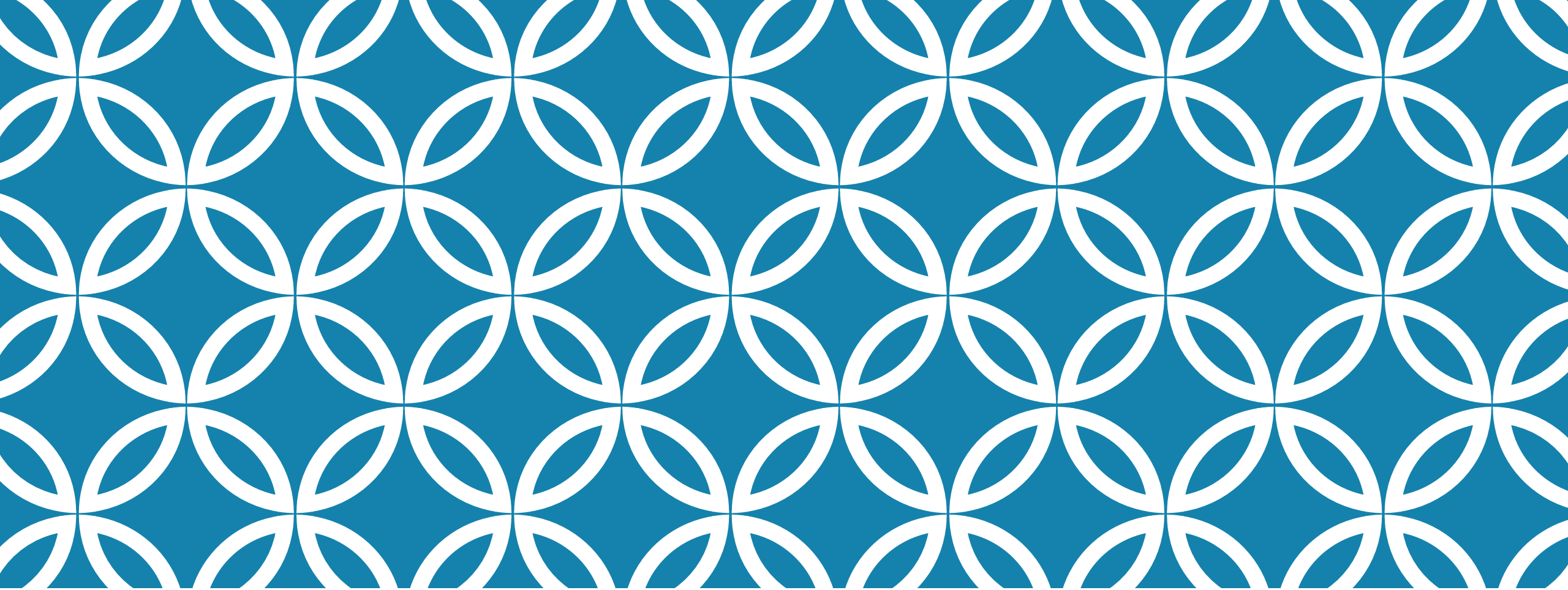
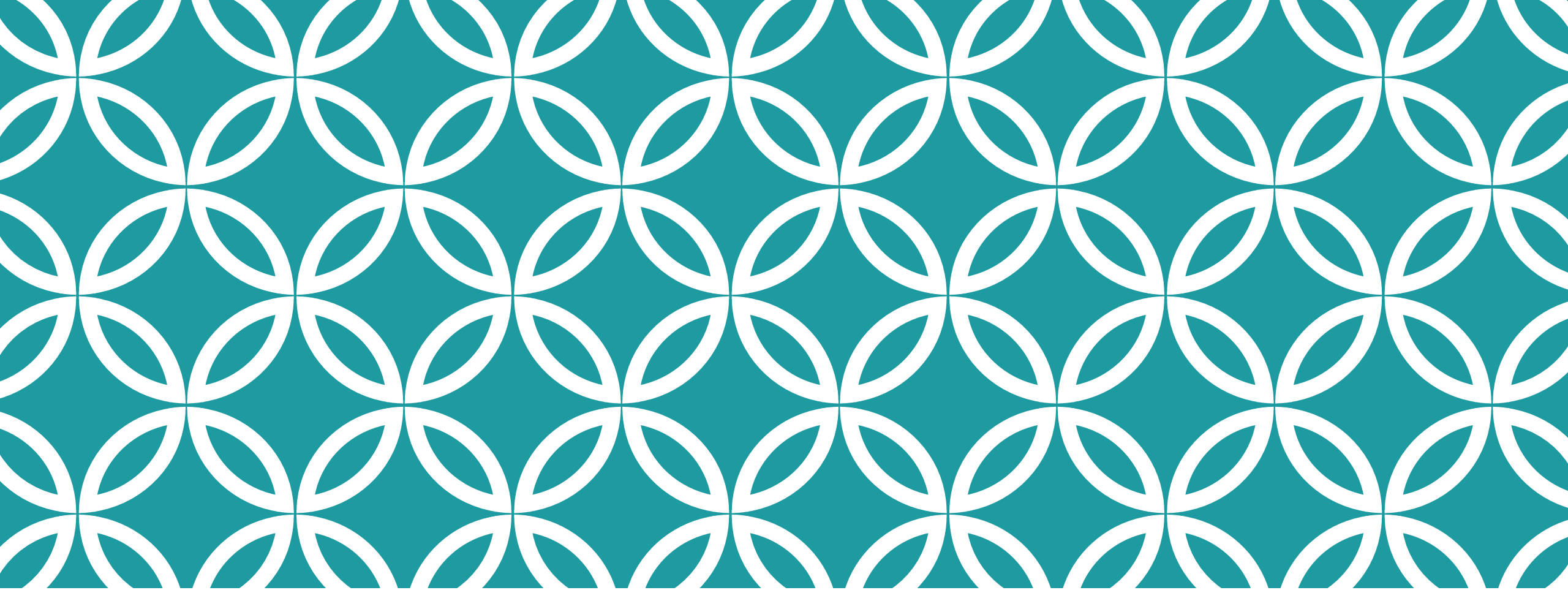


ARITHMETIC OPERATIONS IN INTEL 8086

LECTURE OBJECTIVES

- Be familiar with the syntax and usage of the ADD, INC, DEC, and SUB instructions
- Understand how arithmetic instructions affect the CPU status flags
- Understand and use the MUL, IMUL, DIV, and IDIV instructions
- Know how to perform sign extension of operands in division operations



WHAT'S NEXT

Integer Addition & Subtraction
Integer Multiplication & Division

WHAT'S NEXT

Integer Addition and Subtraction

Integer Multiplication and Division

ADDITION AND SUBTRACTION

INC and DEC Instructions

ADD and SUB Instructions

NEG Instruction

Implementing Arithmetic Expressions

Flags Affected by Arithmetic

- Zero
- Sign
- Carry
- Overflow

INC AND DEC INSTRUCTIONS

Add 1, subtract 1 from destination operand

- operand may be register or memory

INC *destination*

- Logic: *destination* ← *destination* + 1

DEC *destination*

- Logic: *destination* ← *destination* – 1

INC AND DEC EXAMPLES

```
.data
myWord  WORD 1000h
myDword DWORD 10000000h
.code
    inc myWord              ; 1001h
    dec myWord              ; 1000h
    inc myDword             ; 10000001h

    mov ax,00FFh
    inc ax                  ; AX = 0100h
    mov ax,00FFh
    inc al                  ; AX = 0000h
```

ADD AND SUB INSTRUCTIONS

- ADD destination, source
  - Logic: *destination* ← *destination* + source
- SUB destination, source
  - Logic: *destination* ← *destination* – source
- Same operand rules as for the MOV instruction

ADD AND SUB EXAMPLES

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code                          ; ---EAX---
   mov eax,var1                ; 00010000h
   add eax,var2                ; 00030000h
   add ax,0FFFFh               ; 0003FFFFh
   add eax,1                   ; 00040000h
   sub ax,1                    ; 0004FFFFh
```

NEG (NEGATE) INSTRUCTION

Reverses the sign of an operand. Operand can be a register or memory operand.

```
.data
valB BYTE -1
valW WORD +32767
.code
   mov al,valB              ; AL = -1
   neg al                   ; AL = +1
   neg valW                 ; valW = -32767
```

Suppose AX contains –32,768 and we apply NEG to it. Will the result be valid?

NEG INSTRUCTION AND THE FLAGS

The processor implements NEG using the following internal operation:

SUB 0,*operand*

Any nonzero operand causes the Carry flag to be set.

```
.data
valB BYTE 1,0
valC SBYTE -128
.code
   neg valB                ; CF = 1, OF = 0
   neg [valB + 1]          ; CF = 0, OF = 0
   neg valC                ; CF = 1, OF = 1
```

IMPLEMENTING ARITHMETIC EXPRESSIONS

HLL compilers translate mathematical expressions into assembly language. You can do it also. For example:

Rval = -Xval + (Yval – Zval)

```
Rval DWORD ?
Xval DWORD 26
Yval DWORD 30
Zval DWORD 40
.code
   mov eax,Xval
   neg eax                  ; EAX = -26
   mov ebx,Yval
   sub ebx,Zval             ; EBX = -10
   add eax,ebx
   mov Rval,eax             ; -36
```

FLAGS AFFECTED BY ARITHMETIC

The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations

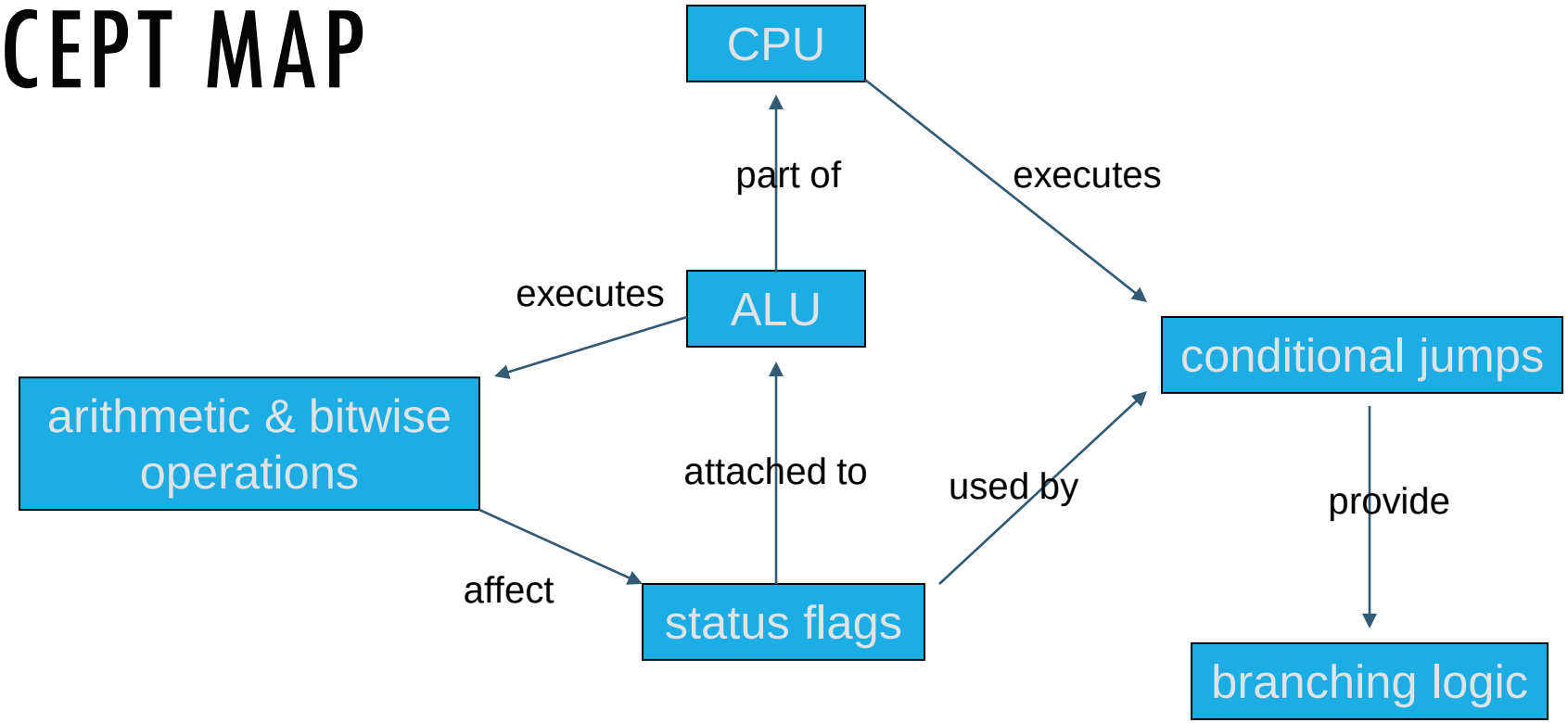
- based on the contents of the destination operand

Essential flags:

- Zero flag – set when destination equals zero
- Sign flag – set when destination is negative
- Carry flag – set when unsigned value is out of range
- Overflow flag – set when signed value is out of range

The MOV instruction never affects the flags.

CONCEPT MAP



You can use diagrams such as these to express the relationships between assembly language concepts.

ZERO FLAG (ZF)

The Zero flag is set when the result of an operation produces zero in the destination operand.

```
mov cx,1
sub cx,1                ; CX = 0, ZF = 1
mov ax,0FFFFh
inc ax                  ; AX = 0, ZF = 1
inc ax                  ; AX = 1, ZF = 0
```

Remember...

- A flag is set when it equals 1.
- A flag is clear when it equals 0.

SIGN FLAG (SF)

The Sign flag is set when the destination operand is negative. The flag is clear when the destination is positive.

```
mov cx,0
sub cx,1                    ; CX = -1, SF = 1
add cx,2                    ; CX = 1, SF = 0
```

The sign flag is a copy of the destination's highest bit:

```
mov al,0
sub al,1                ; AL = 11111111b, SF = 1
add al,2                ; AL = 00000001b, SF = 0
```

SIGNED AND UNSIGNED INTEGERS
A HARDWARE VIEWPOINT

All CPU instructions operate exactly the same on signed and unsigned integers

The CPU cannot distinguish between signed and unsigned integers

YOU, the programmer, are solely responsible for using the correct data type with each instruction

OVERFLOW AND CARRY FLAGS: A HARDWARE VIEWPOINT

How the ADD instruction affects OF and CF:

- CF = (carry out of the MSB)
- OF = CF XOR MSB

How the SUB instruction affects OF and CF:

- CF = INVERT (carry out of the MSB)
- negate the source and add it to the destination
- OF = CF XOR MSB

MSB = Most Significant Bit (high-order bit)
XOR = eXclusive-OR operation
NEG = Negate (same as SUB 0,operand)

CARRY FLAG (CF)

The Carry flag is set when the result of an operation generates an unsigned value that is out of range (too big or too small for the destination operand).

```
mov al,0FFh
add al,1                    ; CF = 1, AL = 00

; Try to go below zero:

mov al,0
sub al,1                    ; CF = 1, AL = FF
```

OVERFLOW FLAG (OF)

The Overflow flag is set when the signed result of an operation is invalid or out of range.

```
; Example 1
mov al,+127
add al,1                ; OF = 1,   AL = ??

; Example 2
mov al,7Fh              ; OF = 1,   AL = 80h
add al,1
```

The two examples are identical at the binary level because 7Fh equals +127. To determine the value of the destination operand, it is often easier to calculate in hexadecimal.

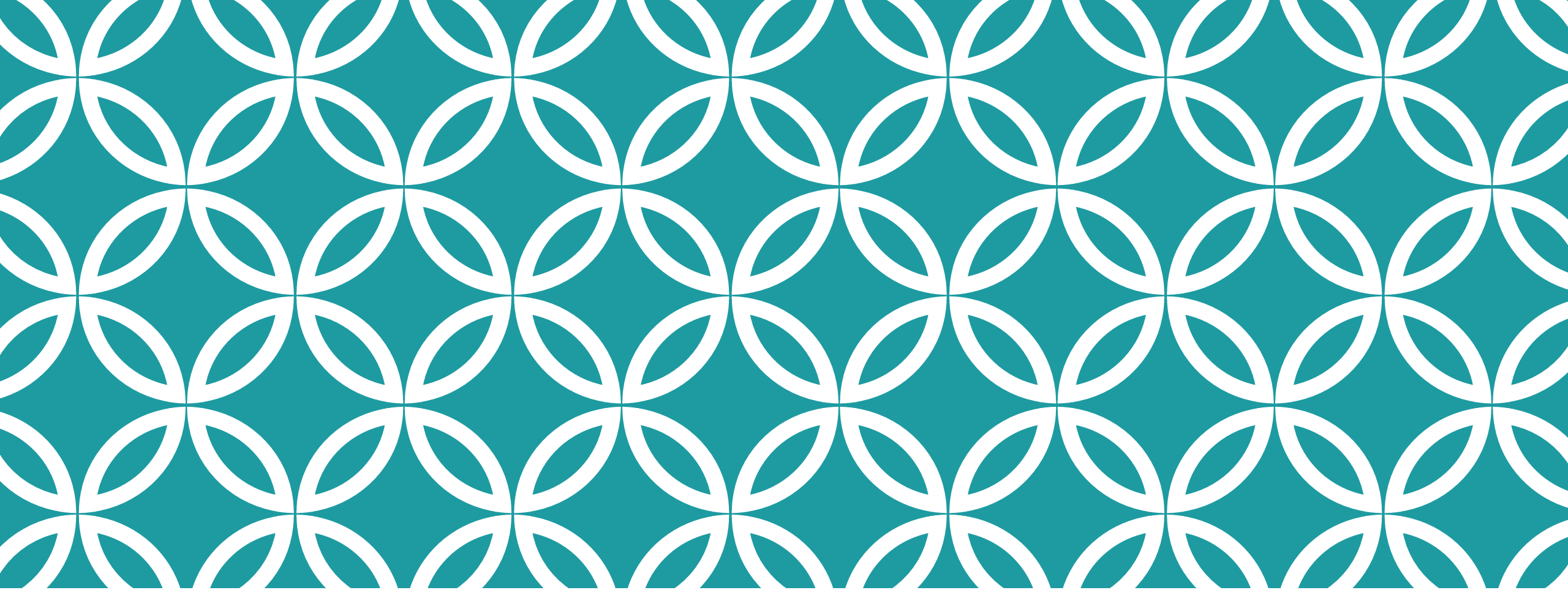
A RULE OF THUMB

When adding two integers, remember that the Overflow flag is only set when . . .

- Two positive operands are added and their sum is negative
- Two negative operands are added and their sum is positive

```
What will be the values of the Overflow flag?
    mov al,80h
    add al,92h              ; OF =          1

    mov al,-2
    add al,+127             ; OF =          0
```



WHAT'S NEXT

Integer Addition & Subtraction

Integer Multiplication & Division

MULTIPLICATION AND DIVISION INSTRUCTIONS

- MUL Instruction
- IMUL Instruction
- DIV Instruction
- Signed Integer Division
- CBW, CWD, CDQ Instructions
- IDIV Instruction
- Implementing Arithmetic Expressions

MUL INSTRUCTION

In 32-bit mode, MUL (unsigned multiply) instruction multiplies an 8-, 16-, or 32-bit operand by either AL, AX, or EAX.

The instruction formats are:

```
MUL r/m8
MUL r/m16
MUL r/m32
```

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

MUL EXAMPLES

100h * 2000h, using 16-bit operands:

```
.data
val1 WORD 2000h
val2 WORD 100h
.code
mov ax,val1
mul val2      ; DX:AX = 00200000h, CF=1
```

The Carry flag indicates whether or not the upper half of the product contains significant digits.

12345h * 1000h, using 32-bit operands:

```
mov eax,12345h
mov ebx,1000h
mul ebx       ; EDX:EAX = 0000000012345000h, CF=0
```

IMUL INSTRUCTION

IMUL (signed integer multiply) multiplies an 8-, 16-, or 32-bit signed operand by either AL, AX, or EAX

Preserves the sign of the product by sign-extending it into the upper half of the destination register

Example: multiply 48 * 4, using 8-bit operands:

```
mov  al,48
mov  bl,4
imul bl          ; AX = 00C0h, OF=1
```

OF=1 because AH is not a sign extension of AL.

IMUL EXAMPLES

Multiply 4,823,424 * –423:

```
mov eax,4823424
mov ebx,-423
imul ebx          ; EDX:EAX = FFFFFFFF86635D80h, OF=0
```

OF=0 because EDX is a sign extension of EAX.

DIV INSTRUCTION

The DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit division on unsigned integers

A single operand is supplied (register or memory operand), which is assumed to be the divisor

Instruction formats:

DIV *reg/mem8*
DIV *reg/mem16*
DIV *reg/mem32*

Default Operands:

Dividend	Divisor	Quotient	Remainder
AX	*r/m8*	AL	AH
DX:AX	*r/m16*	AX	DX
EDX:EAX	*r/m32*	EAX	EDX

DIV EXAMPLES

Divide 8003h by 100h, using 16-bit operands:

```
mov dx,0                ; clear dividend, high
mov ax,8003h            ; dividend, low
mov cx,100h             ; divisor
div cx                  ; AX = 0080h, DX = 3
```

Same division, using 32-bit operands:

```
mov edx,0               ; clear dividend, high
mov eax,8003h           ; dividend, low
mov ecx,100h            ; divisor
div ecx                 ; EAX = 00000080h, DX = 3
```

UNSIGNED DIVISION EXAMPLE

Before
EDX: 00 00 00 00
EAX: 00 00 00 64
EBX: 00 00 00 0D

Instruction
```
div   ebx    ; 100/13
```

After
EDX: 00000009
EAX: 00000007

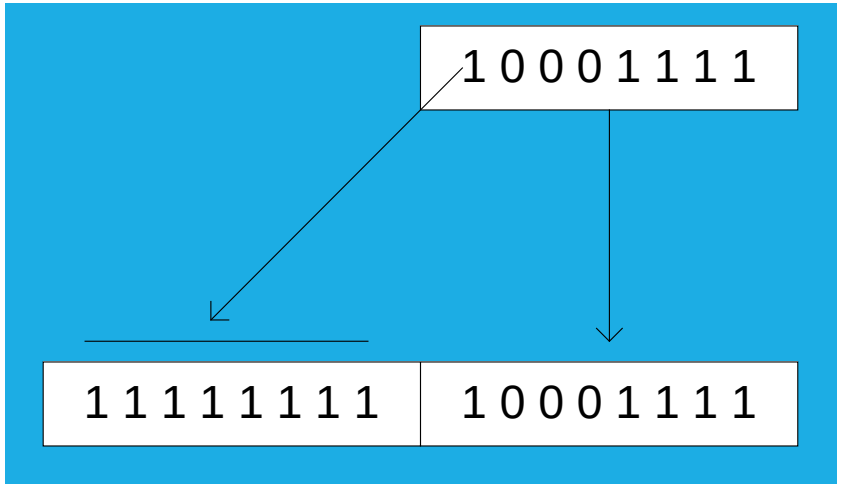
100 = 7 * 13 + 9

SIGNED INTEGER DIVISION (IDIV)

Signed integers must be sign-extended before division takes place

- fill high byte/word/doubleword with a copy of the low byte/word/doubleword's sign bit

For example, the high byte contains a copy of the sign bit from the low byte:



CBW, CWD, CDQ INSTRUCTIONS

The CBW, CWD, and CDQ instructions provide important sign-extension operations:

- CBW (convert byte to word) extends AL into AH
- CWD (convert word to doubleword) extends AX into DX
- CDQ (convert doubleword to quadword) extends EAX into EDX

Example:

```
.data
dwordVal SDWORD -101   ; FFFFFF9Bh
.code
mov eax,dwordVal
cdq                            ; EDX:EAX = FFFFFFFFFFFFF9Bh
```

IDIV INSTRUCTION

IDIV (signed divide) performs signed integer division

Same syntax and operands as DIV instruction

Example: 8-bit division of –48 by 5

```
mov  al,-48
cbw                ; extend AL into AH
mov  bl,5
idiv bl            ; AL = -9,  AH = -3
```

IDIV EXAMPLES

Example: 16-bit division of –48 by 5

```
mov  ax,-48
cwd                ; extend AX into DX
mov  bx,5
idiv bx            ; AX = -9,  DX = -3
```

Example: 32-bit division of –48 by 5

```
mov  eax,-48
cdq                ; extend EAX into EDX
mov  ebx,5
idiv ebx           ; EAX = -9,  EDX = -3
```

SIGNED DIVISION EXAMPLE

Before
EDX: FF FF FF FF
EAX: FF FF FF 9C
ECX: 00 00 00 0D

Instruction
```
idiv  ecx  ; -100/13
```

After
EDX: FFFFFFF7
EAX: FFFFFFF9

–100 = (–7) * 13 + (–9)

UNSIGNED ARITHMETIC EXPRESSIONS

Some good reasons to learn how to implement integer expressions:

- Learn how do compilers do it
- Test your understanding of MUL, IMUL, DIV, IDIV
- Check for overflow (Carry and Overflow flags)

Example: `var4 = (var1 + var2) * var3`

```
; Assume unsigned operands
mov  eax,var1
add  eax,var2        ; EAX = var1 + var2
mul  var3            ; EAX = EAX * var3
jc   TooBig          ; check for carry
mov  var4,eax        ; save product
```

SIGNED ARITHMETIC EXPRESSIONS (1 OF 2)

Example: `eax = (-var1 * var2) + var3`

```
mov  eax,var1
neg  eax
imul var2
jo   TooBig          ; check for overflow
add  eax,var3
jo   TooBig          ; check for overflow
```

Example: `var4 = (var1 * 5) / (var2 – 3)`

```
mov  eax,var1          ; left side
mov  ebx,5
imul ebx               ; EDX:EAX = product
mov  ebx,var2          ; right side
sub  ebx,3
idiv ebx               ; EAX = quotient
mov  var4,eax
```

SIGNED ARITHMETIC EXPRESSIONS (2 OF 2)

Example: `var4 = (var1 * -5) / (-var2 % var3);`

```
mov  eax,var2          ; begin right side
neg  eax
cdq                    ; sign-extend dividend
idiv var3              ; EDX = remainder
mov  ebx,edx           ; EBX = right side
mov  eax,-5            ; begin left side
imul var1              ; EDX:EAX = left side
idiv ebx               ; final division
mov  var4,eax          ; quotient
```

Sometimes it's easiest to calculate the right-hand term of an expression first.

SUMMARY

Addition & Subtraction

- INC and DEC
- ADD and SUB
- FLAGS affected: Zero, Sign, Carry, Overflow

Multiplication & Division

- MUL/IMUL and DIV/IDIV
- Sign Extension: CBW, CWD, CDQ
- FLAGS affected: Zero, Sign, Carry, Overflow