x86 Assembly Programming Part 3

EECE416 uC

Charles Kim
Howard University

Resources:

Intel 80386 Programmers Reference Manual
Essentials of 80x86 Assembly Language
Introduction to 80x86 Assembly Language Programming

WWW.MWFTR.COM

Last week activity - Review

x = 00 00 00 23 → eax
y = 00 00 00 2F
z = 00 00 00 1A → ebx

add eax, y ;

```
    1
  00 00 00 23
  00 00 00 2F +
  ------------
  00 00 00 52
```

15+3=18 → 0X12

CF=0 OF=0

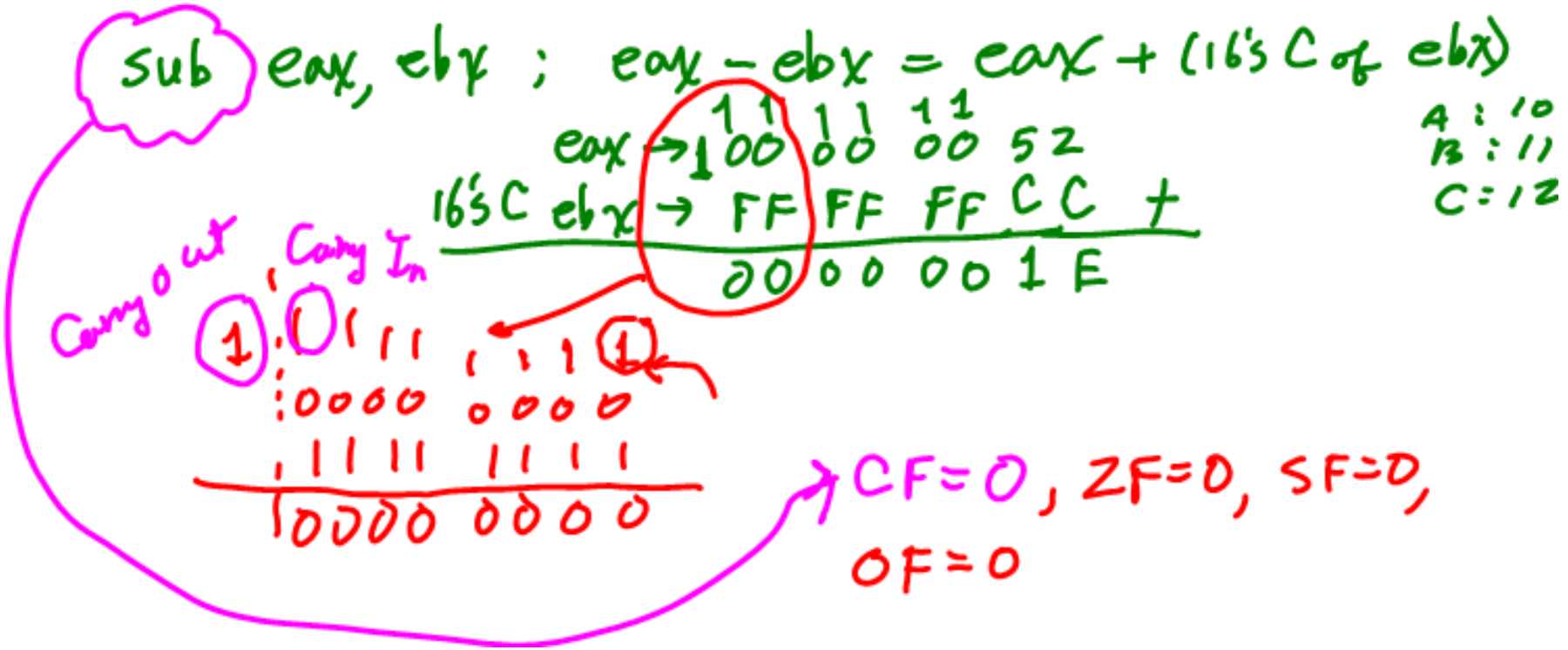
SF=0 ZF=0

add ebx, ebx ;

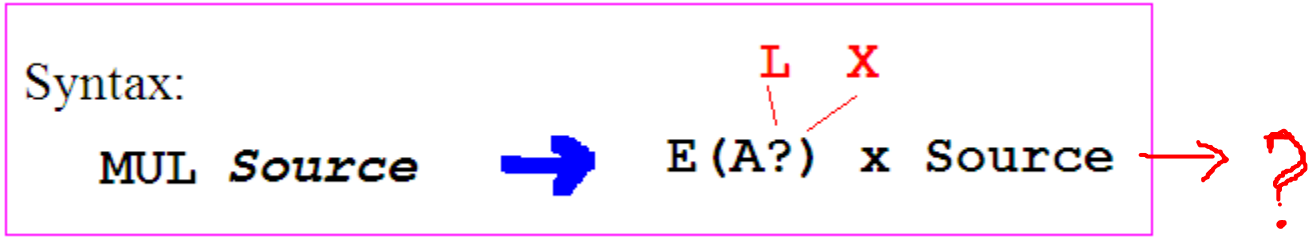
```
     1
  00 00 00 1A
  00 00 00 1A
  ------------
  00 00 00 34
```

10+10=20 → 0X14

Last week activity - Review

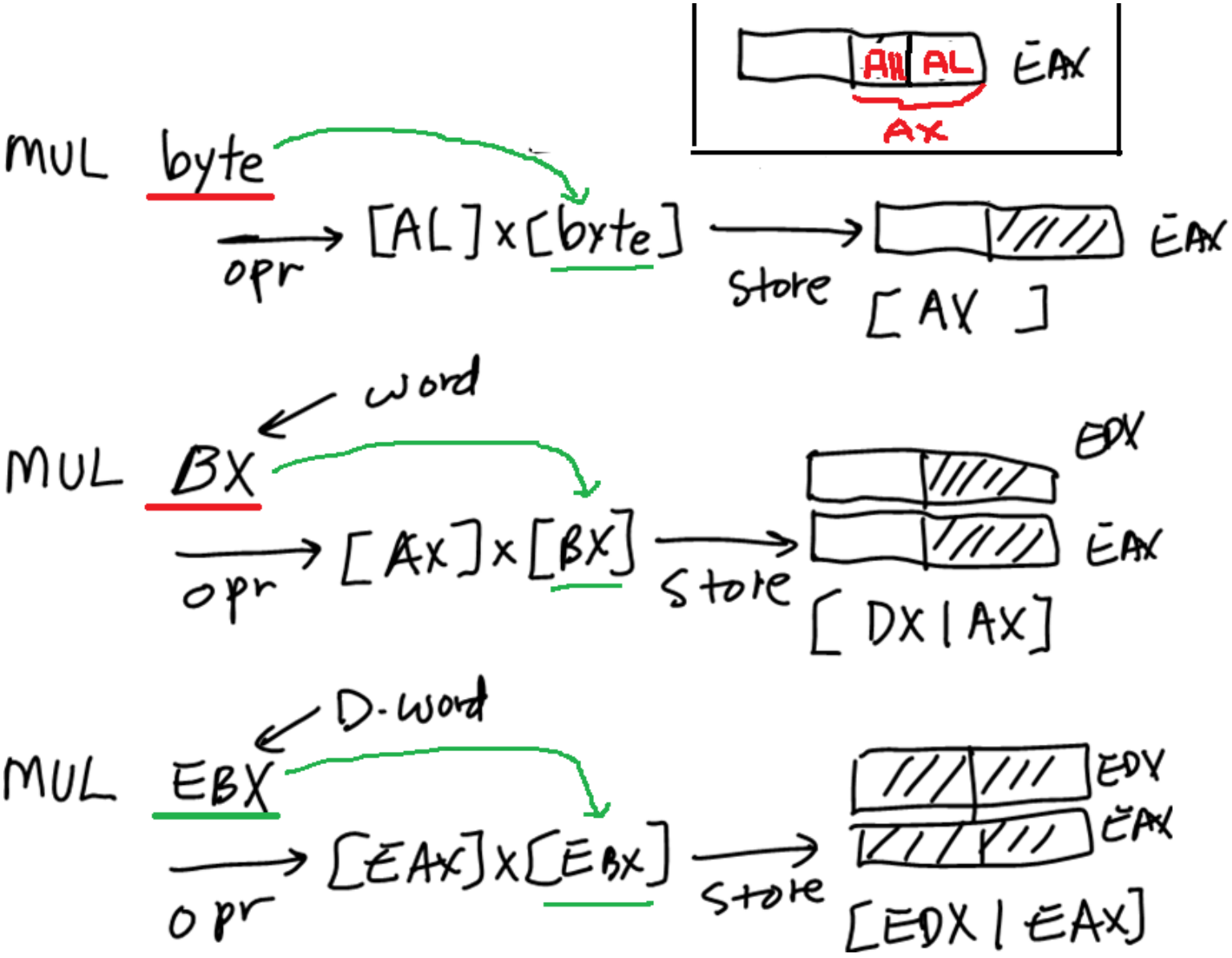


Multiplication Instruction - MUL

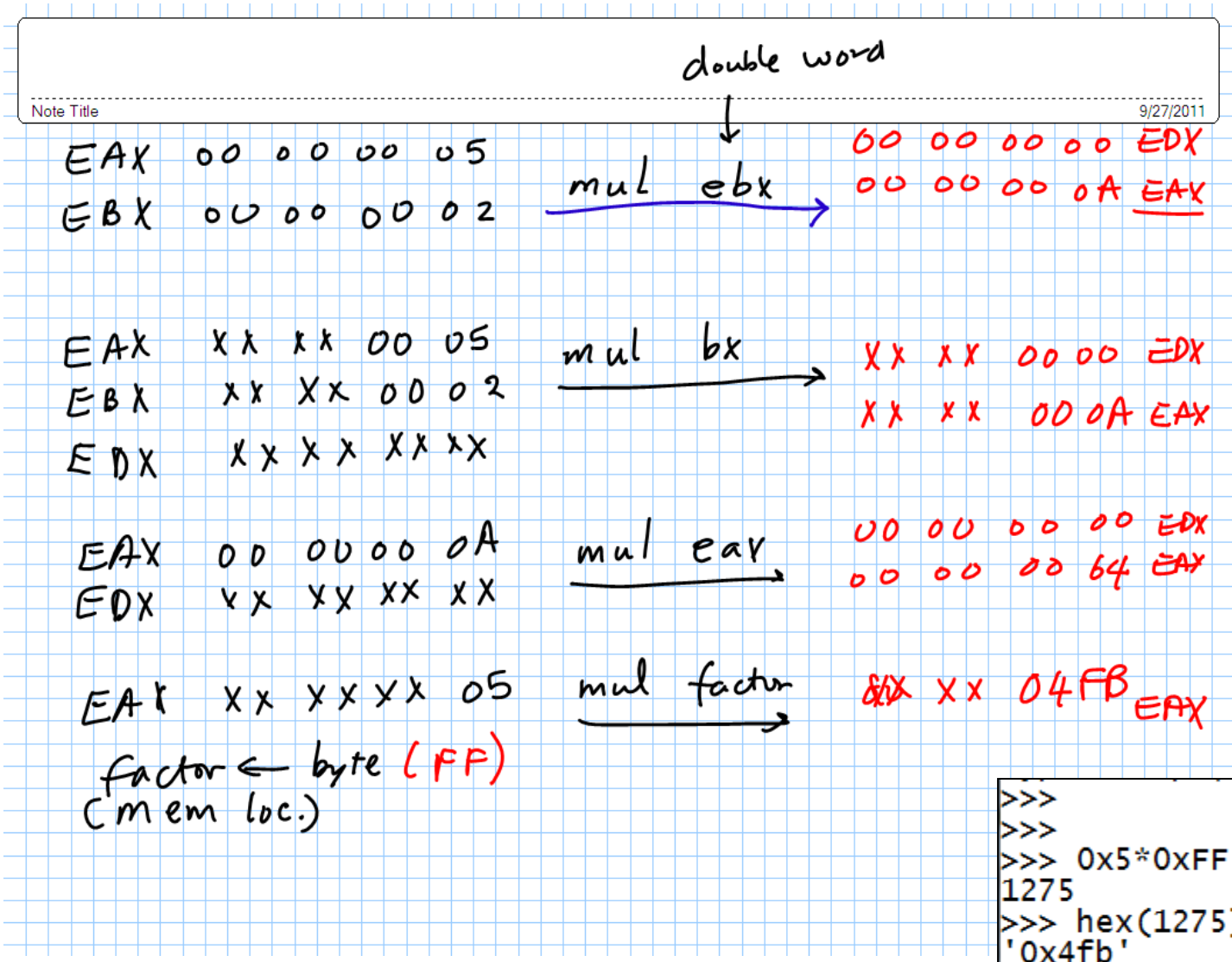


- MUL (Unsigned Integer Multiply)
  - performs an unsigned multiplication of the source operand and the accumulator [(E)AX].
  - **If the source is a byte**
    - the processor multiplies it by the contents of AL and returns the double-length result to AH and AL (Concatenated) i.e, AX.
  - **If the source operand is a word**
    - the processor multiplies it by the contents of AX and returns the double-length result to DX and AX concatenated.
  - **If the source operand is a double-word**
    - the processor multiplies it by the contents of EAX and returns the 64-bit result in EDX and EAX (Concatenated). MUL sets CF and OF when the upper half of the result is nonzero; otherwise, they are cleared.
  - Operand **cannot** be immediate

MUL Opr/Store Summary



MUL - Exercise




```
>>>
>>>
>>> 0x5*0xFF
1275
>>> hex(1275)
'0x4fb'
>>>
```

IMUL (Signed Integer Multiply)

- performs a signed multiplication operation. IMUL has three variations:
  - 1. An **one-operand form.** The operand may be a byte, word, or doubleword located in memory or in a general register. This instruction uses EAX and EDX as implicit operands in the same way as the MUL instruction.

```
imul    source
```

  - 2**. A two-operand form. One of the source operands may be in any general register while the other may be either in memory or in a general register. The product replaces the general-register operand.**

```
imul    destination register, source
```

  - **The immediate operand is treated as signed**. If the immediate operand is **a byte, the processor automatically sign-extends to the size of destination before performing the multiplication.**

IMUL

EAX 00 00 00 05
EBX 00 00 00 02
imul ebx →
EDX 00 00 00 00
EAX 00 00 00 0A

EAX XX XX 00 05
EBX XX XX 00 02
imul bx →
EDX XX XX 00 00
EAX XX XX 00 0A

EAX XX XX XX 05
factor (FF)
imul factor →
EAX XX XX FF FB

EBX 00 00 00 0A
imul ebx, 10 →
EBX 00 00 00 64

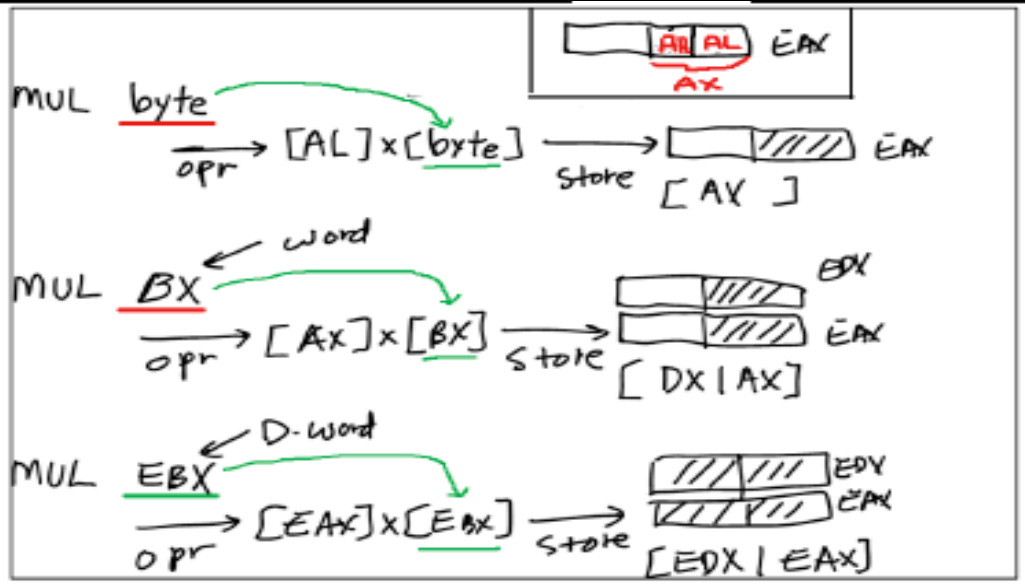
AX: ?? 05
byte at *factor*: FF

-1
imul factor → -5
-5

AX | FF | FB

CF, OF 0

MUL & IMUL Exercise



For each part of the problem, find the requested "after "values for the registers after the "Instruction" is executed.

(1)Before: `[EAX]=FF FF FF E4`
`[EBX]=00 00 00 02`
Instruction: `IMUL EBX`

After: `[EAX]=` _____

`[EDX]=` _____

(2) Before: `[EAX]=00 00 FF FF`
`[EDX]=FF FF 00 02`
Instruction: `IMUL AX`

After: `[EAX]=` _____

(3) Before: `[EAX]=00 00 FF 0F`
`[EBX]=FF FF 1C 02`
`[EDX]=FF FF 00 02`
Instruction: `MUL BH`

After: `[EAX]=` _____

(4) Before: `[EAX]=00 00 00 17`
`[ECX]=00 00 00 B2`
`[EDX]=FF FF 00 02`
Instruction: `MUL ECX`

After: `[EAX]=` _____

`[EDX]=` _____

Division Instruction

Syntax:

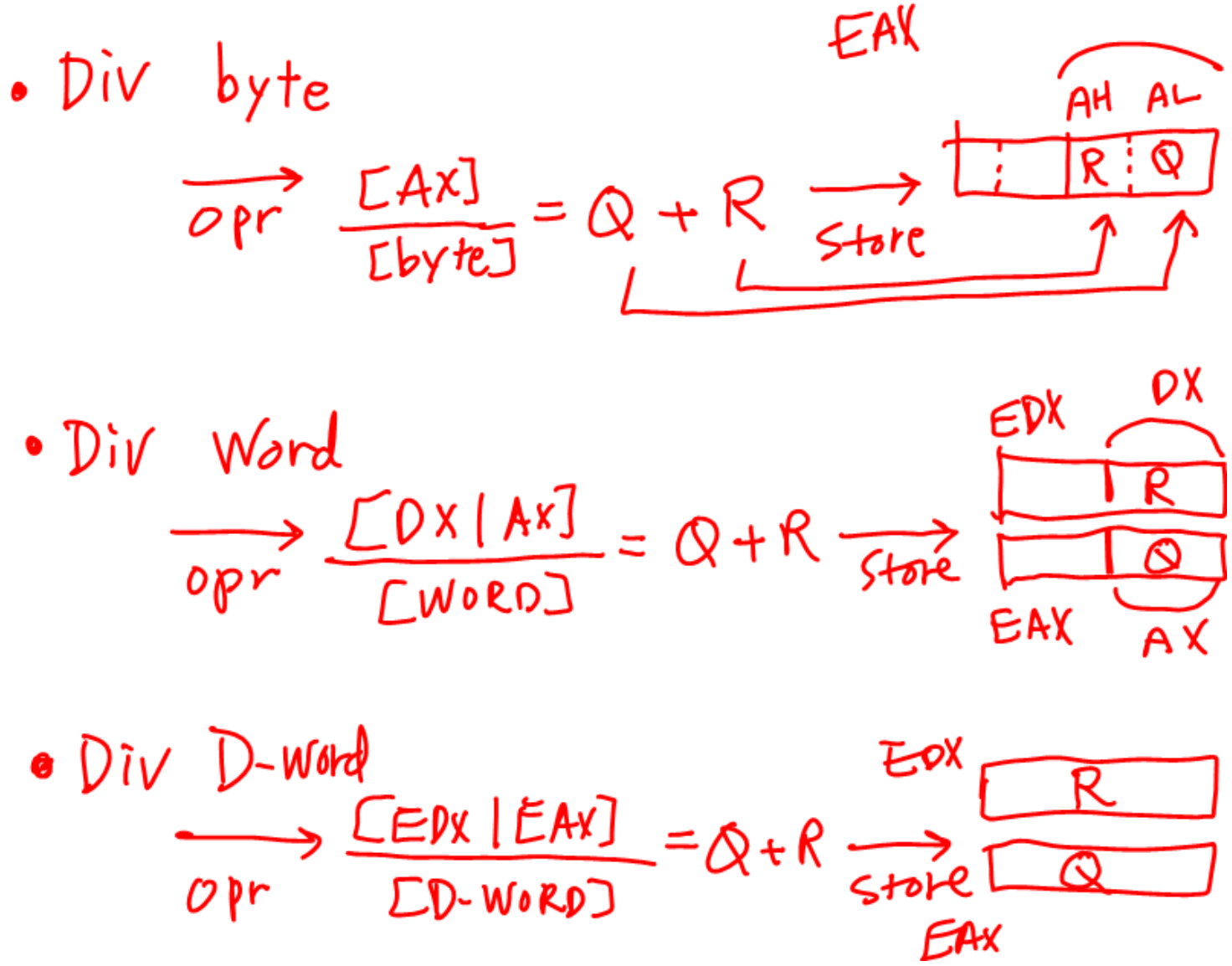
DIV Divider → E(A?)/Divider (Q R)

- DIV (Unsigned Integer Divide)
  - performs an unsigned division of the accumulator by the source operand.
  - The dividend (the accumulator) is twice the size of the divisor (the source operand)
  -

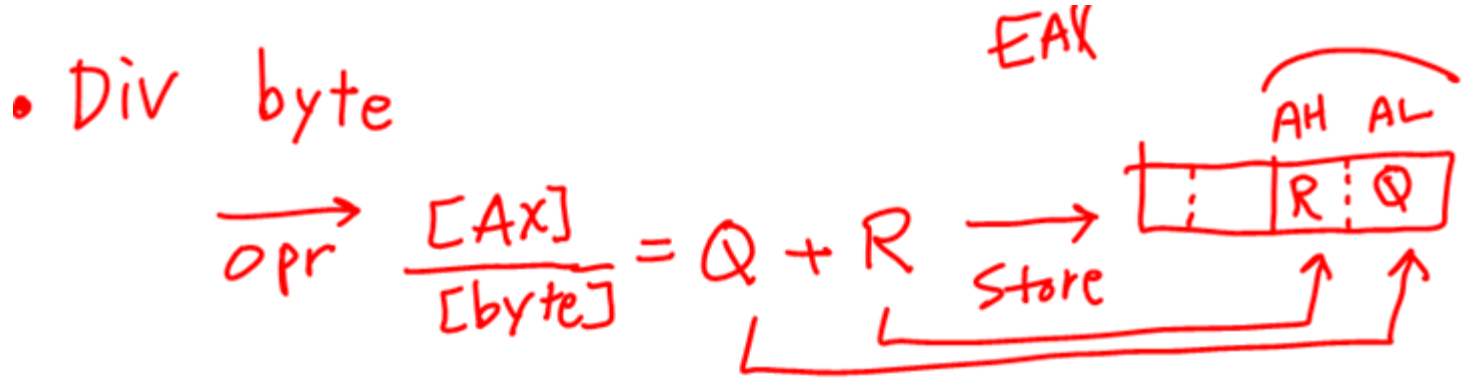
Size of Source Operand (divisor)	Dividend	Quotient	Remainder
Byte	AX	AL	AH
Word	DX:AX	AX	DX
Doubleword	EDX:EAX	EAX	EDX

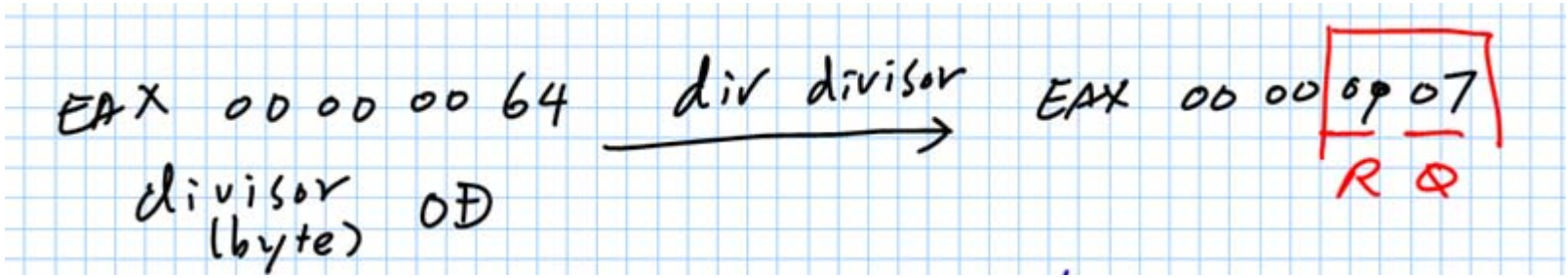
- IDIV (Signed Integer Divide)
  - performs a signed division of the accumulator by the source operand.
  - uses the same registers as the DIV instruction

DIV opr/store summary



DIV & IDIV (Q and R): BYTE divisor

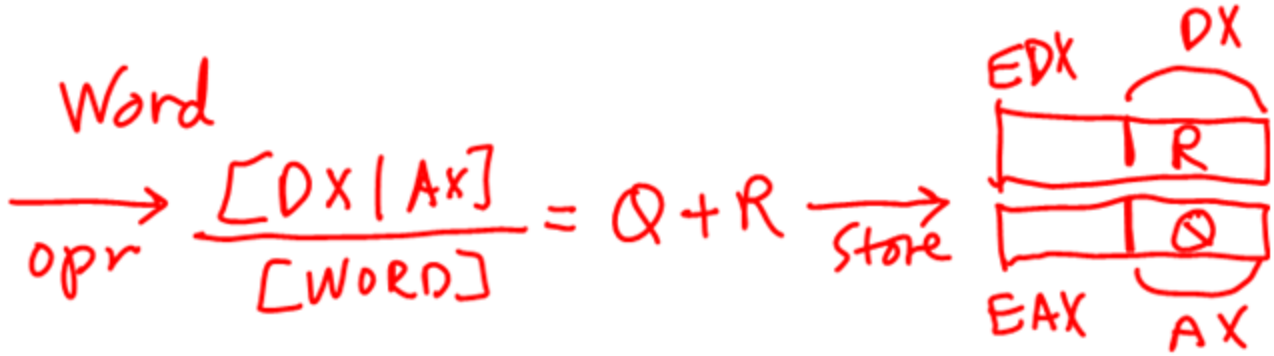




- **0x0064 → 100**
- **0x0D→13**
- **100=Q*13+R =7*13+9**

DIV & IDIV (Q and R) – WORD divisor

- Div Word

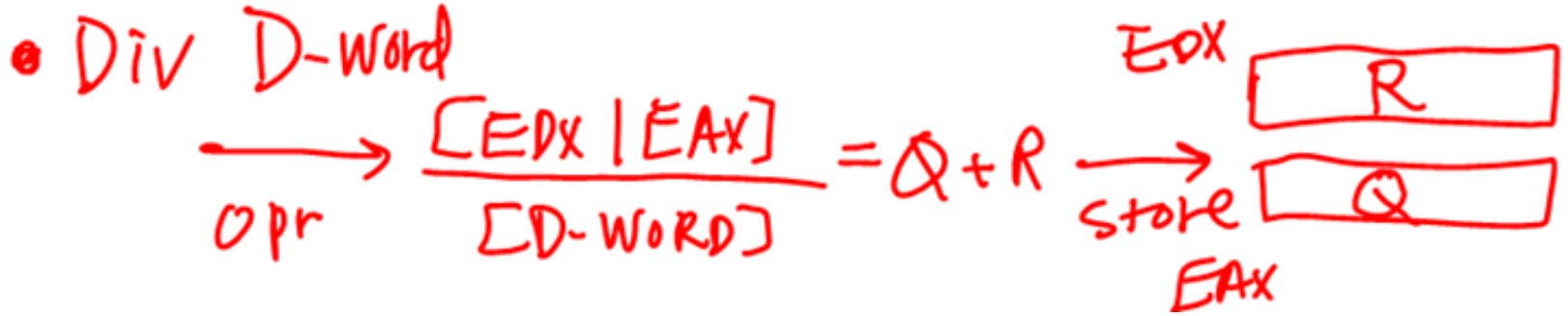


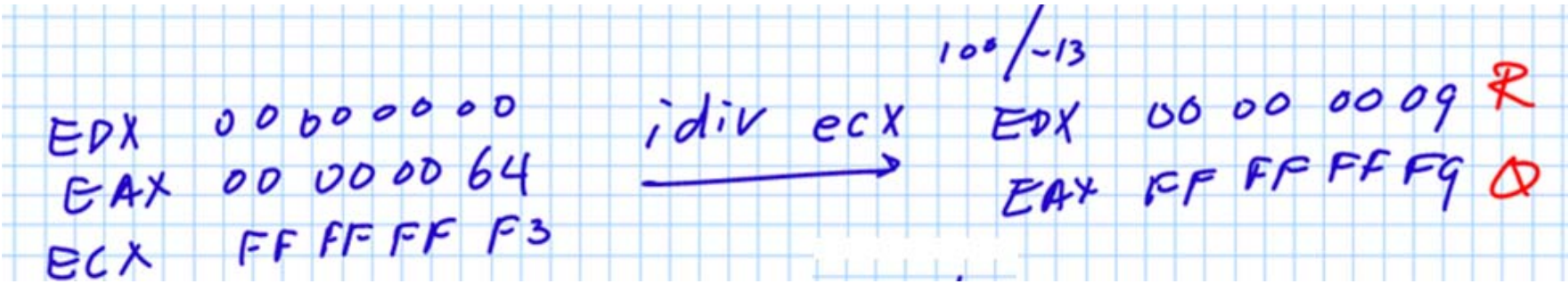
EDX 00 00 00 00
EAX 00 00 00 64
EBX 00 00 00 0D

div bx

100/13
EDX 0000 00 09 R
EAX 00 00 00 07 Q

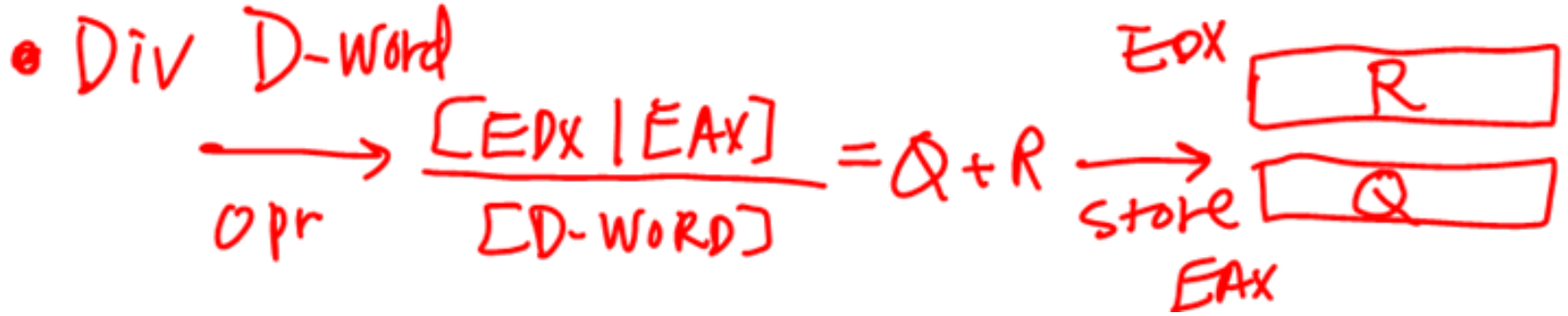
DIV & IDIV (Q and R): DWORD divisor -1

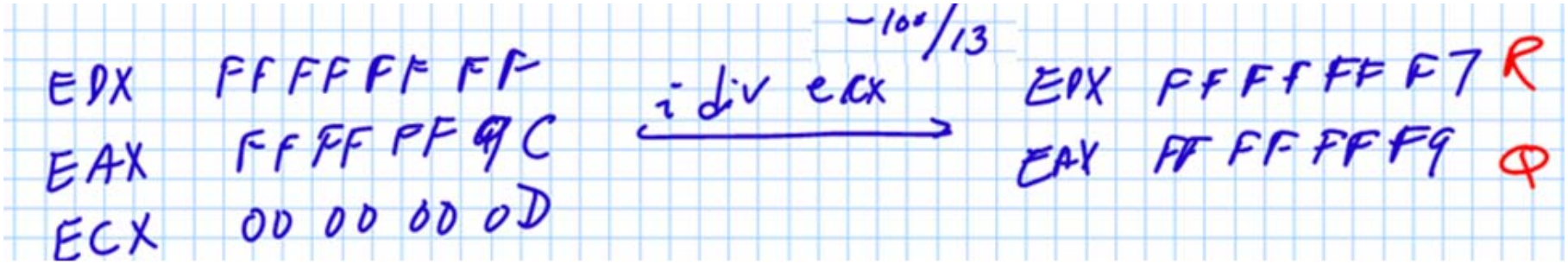




- 0x0000000000000064 → 100
- 0xFFFFFFF3 → - 0xD→ - 13
- 100=Q*(-13)+R = (-7)*(-13)+9
- -7 → 0xFFFFFFF9 9→ 0x00000009

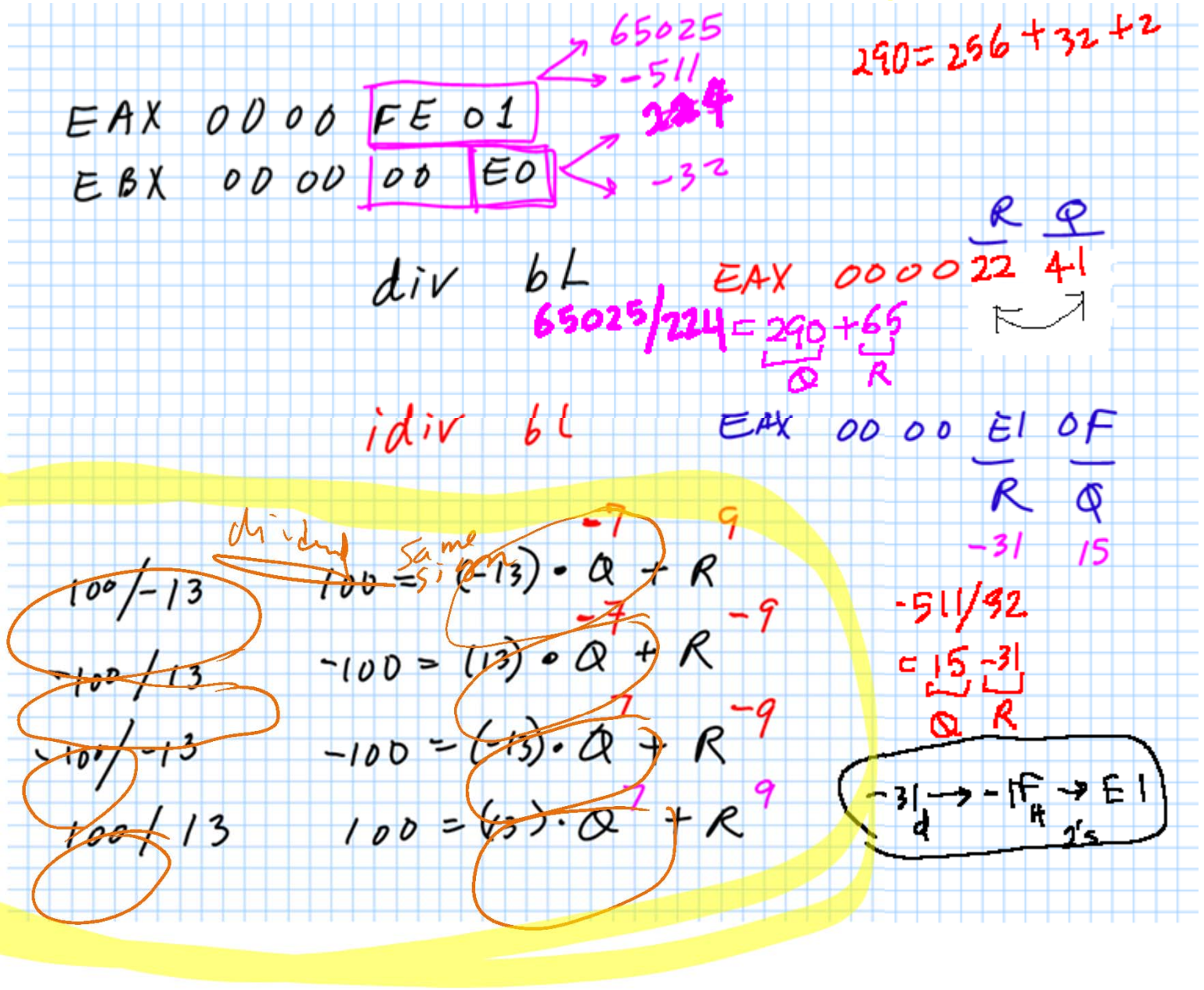
DIV & IDIV (Q and R): DWORD divisor -2





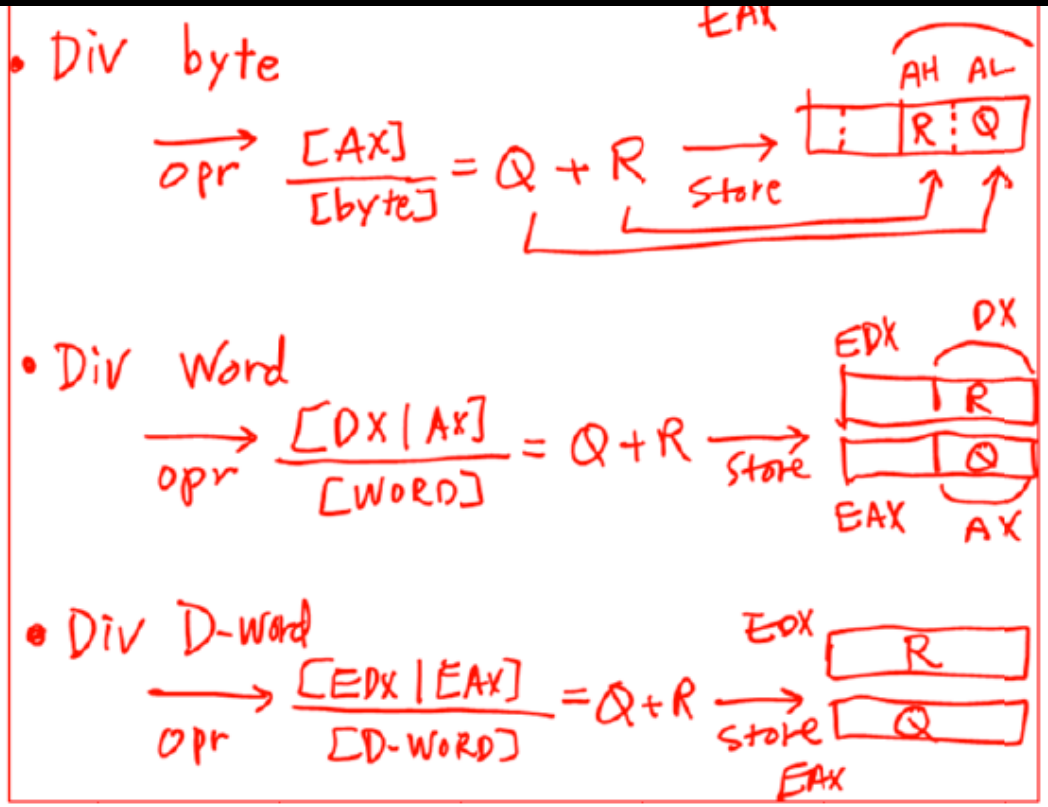
- **0xFFFFFFFFFFFFFF9C → - 0x0000000000000064 → -100**
- **0x0000000D → 13**
- **-100=Q*(13)+R = (-7)*(13)+(-9)**
- **Q=-7 → 0xFFFFFFF9 R=-9→ 0xFFFFFFF7**

DIV vs IDIV: Summary



DIV & IDIV Exercise

- [1]
- 0x9A →154
- 154=10*15 +4
- Q→ 0x0A
- R→0x04



For each part of the problem, assume the "before" values when the given instruction is executed. Find the requested "after "values for the registers when the "Instruction" is executed.

(1) BEFORE:
`[EDX]=00000000, [EAX]=0000009A,`
`[EBX]=0000000F`
Instruction: `IDIV EBX`
AFTER: `[EDX]=`
`[EAX]=`

(4) BEFORE:
`[EDX]=FFFFFFFF, [EAX]=FFFFFF9A,`
`[ECX]=FFFFFFC7`
Instruction: `IDIV ECX`
AFTER: `[EDX]=`
`[EAX]=_____`

Manual Execution with IMUL and IDIV

```
.586
.MODEL FLAT
.STACK  4096

.DATA
TC    DWORD  32     ;
TF    DWORD  ?      ;

.CODE
main      PROC
          mov     eax, TC
          imul    eax,9
          add     eax,2
          mov     ebx,5
          cdq
          idiv    ebx
          add     eax,32
          mov     TF, eax
          mov     eax, 0
          ret

main      ENDP
END
```

Integer

$\frac{5}{2} \Rightarrow 2$

Round off → (3)

$\frac{5+1}{2} \Rightarrow 3$

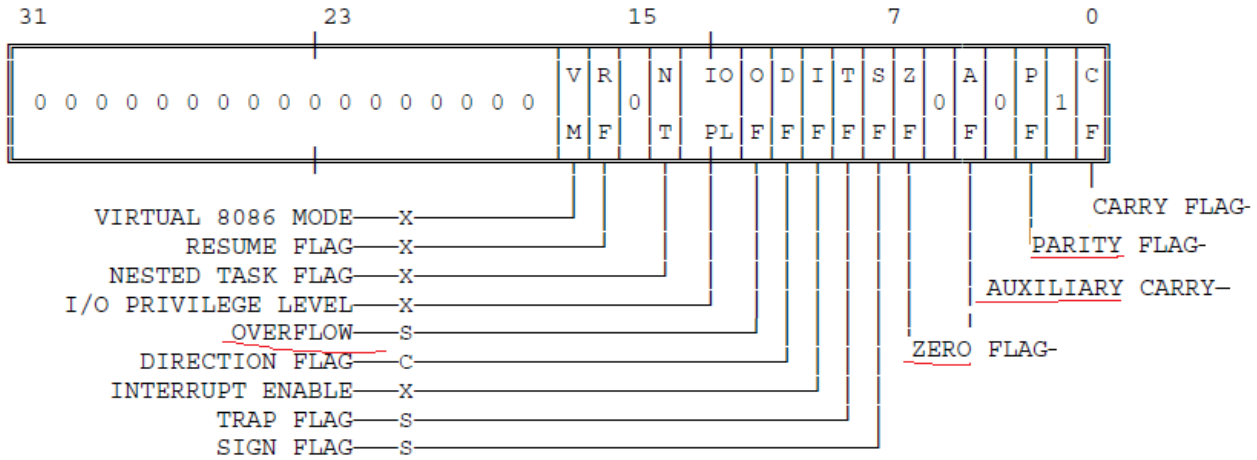
$\frac{1}{2}$ of divisor

- 32→ 0x20
- 32*9=288 →0x120
- 0x120+2→ 0x122
- 0x122/0x5= 0x3A (ignore Remainder)
- 0x3A+0x20 →0x5A→90
- cf:

```
>>> 32*9/5. + 32
89.599999999999994
```

Boolean Operation Instruction

- AND, OR, XOR, and NOT
- NOT (Not)
  - inverts the bits in the specified operand to form a one's complement of the operand.
  - a unary operation that uses a single operand in a register or memory.
  - has no effect on the flags.
- AND: logical operation of "and“
- OR: Logical operation of “(inclusive)or”
- XOR: Logical operation of "exclusive or".
- AND, OR, XOR clear OF and CF, leave AF undefined, and update SF, ZF, and PF.



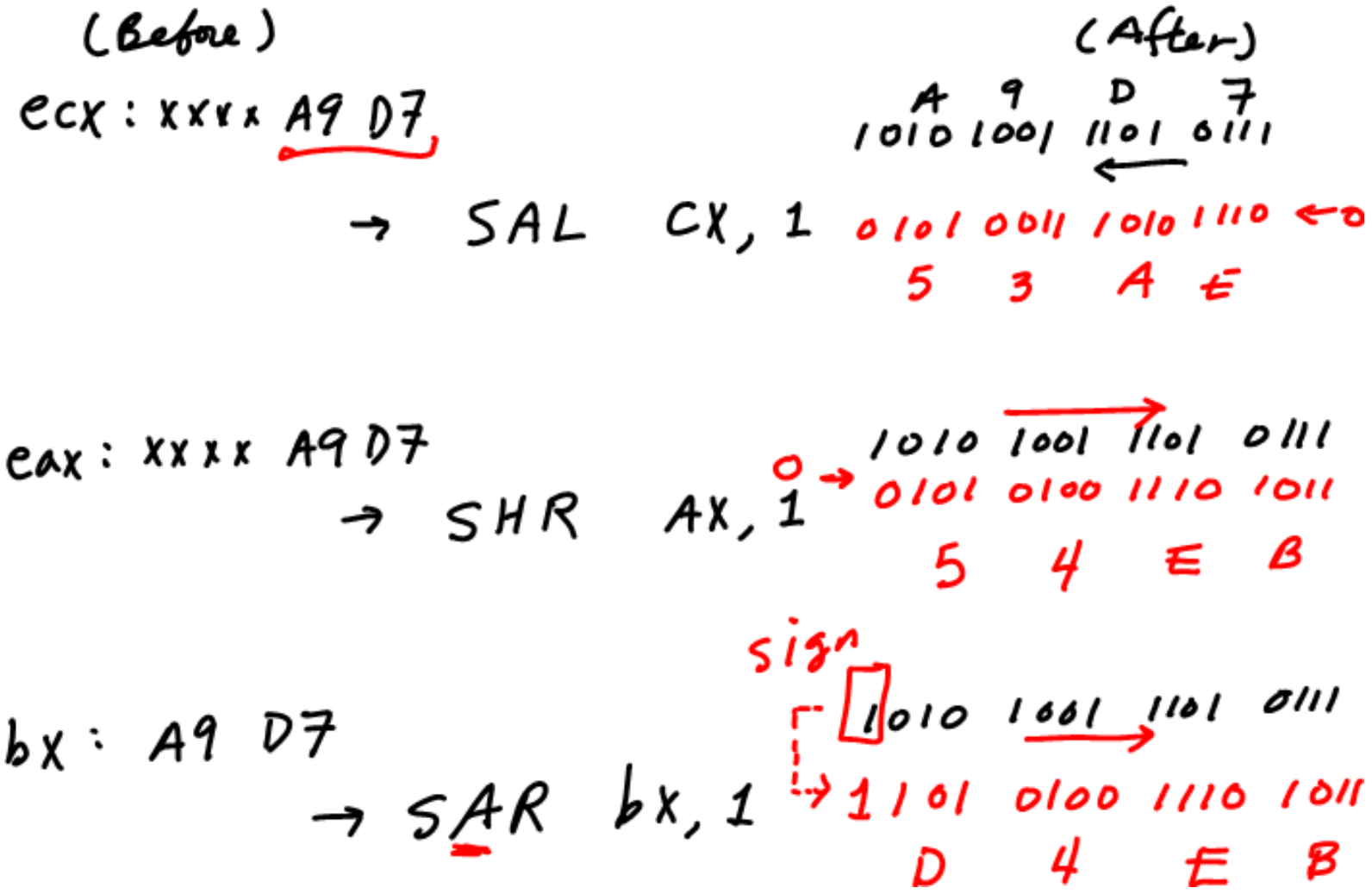
Shift Instructions

- The bits in bytes, words, and doublewords may be shifted arithmetically or logically, up to 31 places.
- Specification of the count of shift
  - Implicitly as a single shift
  - Immediate value
  - Value contained in the CL (lower order 5 bits)
- **CF** always contains the value of the last bit shifted out of the destination operand.
- In a single-bit shift, **OF** is set if the value of the high-order (**sign**) bit was changed by the operation. Otherwise, OF is cleared.
- The shift instructions provide a convenient way to **accomplish division or multiplication by binary power**.

SAL, SAR, SHL, SHR

- SAL (Shift Arithmetic Left) :
  - Fill the right-most bit with 0
- SAR (Shift Arithmetic Right):
  - Fill the left-most bit with the sign bit of the operand
- SHL (Shift Logical Left):
  - Fill the left-most bit with 0
- SHR (Shift Logical Right):
  - Fill the right-most bit with 0;
  - Fill CF with the bit shifted out of the right.

Example


(Before)
ecx : xxxx A9 D7
→ SAL CX, 1
(After)
A 9 D 7
1010 1001 1101 0111
0101 0011 1010 1110 ←0
5 3 A E
eax : xxxx A9 D7
→ SHR AX, 1
0 →
1010 1001 1101 0111
0101 0100 1110 1011
5 4 E B
bx : A9 D7
→ SAR bx, 1
sign
1010 1001 1101 0111
1101 0100 1110 1011
D 4 E B

Example

dx : A9 D7

→ SHR dx, 4

1010 1001 1101 0111

0 → 0000 1010 1001 1101

0 A 9 D

Ax : A9 D7

cL : 04 → SAR ax, cL

1010 1001 1101 0111

① 1101 0100 1110 1011

② 1110 1010 0111 0101

③ 1111 0101 0011 1010

④ 1111 1010 1001 1101

F A 9 D

Next topic ---- Cmp and Jump

JUMPS (flags remain unchanged)			
Name	**Comment**	**Code**	**Operation**
CALL	Call subroutine	CALL Proc	
JMP	Jump	JMP Dest	
JE	Jump if Equal	JE Dest	(≡ JZ)
JZ	Jump if Zero	JZ Dest	(≡ JE)
JCXZ	Jump if CX Zero	JCXZ Dest	
JP	Jump if Parity (Parity Even)	JP Dest	(≡ JPE)
JPE	Jump if Parity Even	JPE Dest	(≡ JP)

Name	**Comment**	**Code**	**Operation**
RET	Return from subroutine	RET	
JNE	Jump if not Equal	JNE Dest	(≡ JNZ)
JNZ	Jump if not Zero	JNZ Dest	(≡ JNE)
JECXZ	Jump if ECX Zero	JECXZ Dest	
JNP	Jump if no Parity (Parity Odd)	JNP Dest	(≡ JPO)
JPO	Jump if Parity Odd	JPO Dest	(≡ JNP)

JUMPS Unsigned (Cardinal)			
JA	Jump if Above	JA Dest	(≡ JNBE)
JAE	Jump if Above or Equal	JAE Dest	(≡ JNB ≡ JNC)
JB	Jump if Below	JB Dest	(≡ JNAE ≡ JC)
JBE	Jump if Below or Equal	JBE Dest	(≡ JNA)
JNA	Jump if not Above	JNA Dest	(≡ JBE)
JNAE	Jump if not Above or Equal	JNAE Dest	(≡ JB ≡ JC)
JNB	Jump if not Below	JNB Dest	(≡ JAE ≡ JNC)
JNBE	Jump if not Below or Equal	JNBE Dest	(≡ JA)
JC	Jump if Carry	JC Dest	
JNC	Jump if no Carry	JNC Dest	

JUMPS Signed (Integer)			
JG	Jump if Greater	JG Dest	(≡ JNLE)
JGE	Jump if Greater or Equal	JGE Dest	(≡ JNL)
JL	Jump if Less	JL Dest	(≡ JNGE)
JLE	Jump if Less or Equal	JLE Dest	(≡ JNG)
JNG	Jump if not Greater	JNG Dest	(≡ JLE)
JNGE	Jump if not Greater or Equal	JNGE Dest	(≡ JL)
JNL	Jump if not Less	JNL Dest	(≡ JGE)
JNLE	Jump if not Less or Equal	JNLE Dest	(≡ JG)
JO	Jump if Overflow	JO Dest	
JNO	Jump if no Overflow	JNO Dest	
JS	Jump if Sign (= negative)	JS Dest	
JNS	Jump if no Sign (= positive)	JNS Dest	

General Registers: