



Smith College

Computer Science

# CSC231 - Assembly

Week #5

Dominique Thiébaud  
dthiebaut@smith.edu

0000	<b>0</b>	<b>0</b>
0001	<b>1</b>	<b>1</b>
0010	<b>2</b>	<b>2</b>
0011	<b>3</b>	<b>3</b>
0100	<b>4</b>	<b>4</b>
0101	<b>5</b>	<b>5</b>
0110	<b>6</b>	<b>6</b>
0111	<b>7</b>	<b>7</b>
1000	<b>8</b>	<b>8</b>
1001	<b>9</b>	<b>9</b>
1010	<b>A</b>	<b>10</b>
1011	<b>B</b>	<b>11</b>
1100	<b>C</b>	<b>12</b>
1101	<b>D</b>	<b>13</b>
1110	<b>E</b>	<b>14</b>
1111	<b>F</b>	<b>15</b>

# Conversion

Super  
Useful!

# Decimal to Binary Conversion

Review  
Shifting

101001 =

$$101001 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$\begin{aligned} 101001 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 32 + 0 + 8 + 0 + 0 + 1 \\ &= 41d \end{aligned}$$

$$\begin{aligned} 101001 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 32 + 0 + 8 + 0 + 0 + 1 \\ &= 41d \end{aligned}$$

$$\frac{101001}{2} = \frac{1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0}{2}$$

$$\begin{aligned}101001 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 32 + 0 + 8 + 0 + 0 + 1 \\ &= 41d\end{aligned}$$

$$\begin{aligned}\frac{101001}{2} &= \frac{1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0}{2} \\ &= \frac{1 \times 2^5}{2} + \frac{0 \times 2^4}{2} + \frac{1 \times 2^3}{2} + \frac{0 \times 2^2}{2} + \frac{0 \times 2^1}{2} + \frac{1 \times 2^0}{2}\end{aligned}$$



$$\begin{aligned}
101001 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 32 + 0 + 8 + 0 + 0 + 1 \\
&= 41d
\end{aligned}$$

$$\begin{aligned}
\frac{101001}{2} &= \frac{1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0}{2} \\
&= \frac{1 \times 2^5}{2} + \frac{0 \times 2^4}{2} + \frac{1 \times 2^3}{2} + \frac{0 \times 2^2}{2} + \frac{0 \times 2^1}{2} + \frac{1 \times 2^0}{2} \\
&= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0
\end{aligned}$$

$$\begin{aligned}
101001 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 32 + 0 + 8 + 0 + 0 + 1 \\
&= 41d
\end{aligned}$$

$$\begin{aligned}
\frac{101001}{2} &= \frac{1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0}{2} \\
&= \frac{1 \times 2^5}{2} + \frac{0 \times 2^4}{2} + \frac{1 \times 2^3}{2} + \frac{0 \times 2^2}{2} + \frac{0 \times 2^1}{2} + \frac{1 \times 2^0}{2} \\
&= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\
&= 10100
\end{aligned}$$

$$\begin{aligned}
101001 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 32 + 0 + 8 + 0 + 0 + 1 \\
&= 41d
\end{aligned}$$

$$\begin{aligned}
\frac{101001}{2} &= \frac{1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0}{2} \\
&= \frac{1 \times 2^5}{2} + \frac{0 \times 2^4}{2} + \frac{1 \times 2^3}{2} + \frac{0 \times 2^2}{2} + \frac{0 \times 2^1}{2} + \frac{1 \times 2^0}{2} \\
&= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\
&= 10100 \\
&= 16 + 0 + 4 + 0 + 0 \\
&= 20d
\end{aligned}$$

**Dividing by the base  
extracts the least  
significant digit**

# Python Program

```
# prompts user for an integer
# decomposes the integer into binary

x = int( input( "> " ) )
binary = ""

while True:
    if x==0:
        break

    remainder = x % 2
    quotient  = x // 2

    if remainder == 0:
        binary = "0" + binary
    else:
        binary = "1" + binary

    print( "%5d = %5d * 2 + %d      quotient=%5d remainder=%d binary=%16s"
           % (x, quotient, remainder, quotient, remainder, binary) )

    x = quotient
```

# Python Program

```
# prompts user for an int
# decomposes the integer
```

```
x = int( input( "> " ) )
binary = ""
```

```
while True:
    if x==0:
        break
```

```
    remainder = x % 2
    quotient = x // 2
```

```
    if remainder == 0:
        binary = "0" + binary
    else:
        binary = "1" + binary
```

```
    print( "%5d = %5d * 2 + %d    quotient=%5d remainder=%d binary=%16s"
           % (x, quotient, remainder, quotient, remainder, binary) )
```

```
    x = quotient
```

```
fiveNumbers35 -- pi@raspberrypiREG: ~ -- ssh -- 88x20
pi@raspberrypiREG: ~
231b@aurora ~/handout $ ./decimal2binary.py
> 12345
12345 = 6172 * 2 + 1    (quotient= 6172 remainder= 1) -- binary= 1
6172 = 3086 * 2 + 0    (quotient= 3086 remainder= 0) -- binary= 01
3086 = 1543 * 2 + 0    (quotient= 1543 remainder= 0) -- binary= 001
1543 = 771 * 2 + 1     (quotient= 771 remainder= 1) -- binary= 1001
771 = 385 * 2 + 1     (quotient= 385 remainder= 1) -- binary= 11001
385 = 192 * 2 + 1     (quotient= 192 remainder= 1) -- binary= 111001
192 = 96 * 2 + 0      (quotient= 96 remainder= 0) -- binary= 0111001
96 = 48 * 2 + 0       (quotient= 48 remainder= 0) -- binary= 00111001
48 = 24 * 2 + 0       (quotient= 24 remainder= 0) -- binary= 000111001
24 = 12 * 2 + 0       (quotient= 12 remainder= 0) -- binary= 0000111001
12 = 6 * 2 + 0        (quotient= 6 remainder= 0) -- binary= 00000111001
6 = 3 * 2 + 0         (quotient= 3 remainder= 0) -- binary= 000000111001
3 = 1 * 2 + 1         (quotient= 1 remainder= 1) -- binary= 1000000111001
1 = 0 * 2 + 1         (quotient= 0 remainder= 1) -- binary= 11000000111001
231b@aurora ~/handout $
```

# Binary to Decimal

$$\begin{aligned}101001 &= 1x2^5 + 0x2^4 + 1x2^3 + 0x2^2 + 0x2^1 + 1x2^0 \\ &= 32 + 0 + 8 + 0 + 0 + 1 \\ &= 41d\end{aligned}$$



# Binary to Hex

1010010 =

$$1010010 = 01010010$$

$$\begin{aligned} 1010010 &= 01010010 \\ &= 0101_2 0010_2 \end{aligned}$$

0000	<b>0</b>
0001	<b>1</b>
0010	<b>2</b>
0011	<b>3</b>
0100	<b>4</b>
0101	<b>5</b>
0110	<b>6</b>
0111	<b>7</b>
1000	<b>8</b>
1001	<b>9</b>
1010	<b>A</b>
1011	<b>B</b>
1100	<b>C</b>
1101	<b>D</b>
1110	<b>E</b>
1111	<b>F</b>

$$\begin{aligned} 1010010 &= 01010010 \\ &= 0101_2 0010_2 \\ &\quad \wedge \end{aligned}$$

0000	<b>0</b>
0001	<b>1</b>
0010	<b>2</b>
0011	<b>3</b>
0100	<b>4</b>
0101	<b>5</b>
0110	<b>6</b>
0111	<b>7</b>
1000	<b>8</b>
1001	<b>9</b>
1010	<b>A</b>
1011	<b>B</b>
1100	<b>C</b>
1101	<b>D</b>
1110	<b>E</b>
1111	<b>F</b>

$$\begin{aligned} 1010010 &= 01010010 \\ &= 0101\ 0010 \\ &= \quad 5 \quad 2 \end{aligned}$$

# Hex to Binary

0000	<b>0</b>
0001	<b>1</b>
0010	<b>2</b>
0011	<b>3</b>
0100	<b>4</b>
0101	<b>5</b>
0110	<b>6</b>
0111	<b>7</b>
1000	<b>8</b>
1001	<b>9</b>
1010	<b>A</b>
1011	<b>B</b>
1100	<b>C</b>
1101	<b>D</b>
1110	<b>E</b>
1111	<b>F</b>

5A1F =



0000	<b>0</b>
0001	<b>1</b>
0010	<b>2</b>
0011	<b>3</b>
0100	<b>4</b>
0101	<b>5</b>
0110	<b>6</b>
0111	<b>7</b>
1000	<b>8</b>
1001	<b>9</b>
1010	<b>A</b>
1011	<b>B</b>
1100	<b>C</b>
1101	<b>D</b>
1110	<b>E</b>
1111	<b>F</b>

5A1F = 0101 1010 0001 1111

# Hex to Decimal

1A2F =

$$1A2F = 1 \times 16^3 + A \times 16^2 + 2 \times 16^1 + F \times 16^0$$
$$=$$

0000	<b>0</b>
0001	<b>1</b>
0010	<b>2</b>
0011	<b>3</b>
0100	<b>4</b>
0101	<b>5</b>
0110	<b>6</b>
0111	<b>7</b>
1000	<b>8</b>
1001	<b>9</b>
1010	<b>A</b>
1011	<b>B</b>
1100	<b>C</b>
1101	<b>D</b>
1110	<b>E</b>
1111	<b>F</b>

$$\begin{aligned} 1A2F &= 1 \times 16^3 + A \times 16^2 + 2 \times 16^1 + F \times 16^0 \\ &= 1 \times 4096 + 10 \times 256 + 2 \times 16 + 15 \times 1 \\ &= \end{aligned}$$

0000	<b>0</b>
0001	<b>1</b>
0010	<b>2</b>
0011	<b>3</b>
0100	<b>4</b>
0101	<b>5</b>
0110	<b>6</b>
0111	<b>7</b>
1000	<b>8</b>
1001	<b>9</b>
1010	<b>A</b>
1011	<b>B</b>
1100	<b>C</b>
1101	<b>D</b>
1110	<b>E</b>
1111	<b>F</b>

$$\begin{aligned} 1A2F &= 1 \times 16^3 + A \times 16^2 + 2 \times 16^1 + F \times 16^0 \\ &= 1 \times 4096 + 10 \times 256 + 2 \times 16 + 15 \times 1 \\ &= 4096 + 2560 + 32 + 15 \\ &= \end{aligned}$$

0000	<b>0</b>
0001	<b>1</b>
0010	<b>2</b>
0011	<b>3</b>
0100	<b>4</b>
0101	<b>5</b>
0110	<b>6</b>
0111	<b>7</b>
1000	<b>8</b>
1001	<b>9</b>
1010	<b>A</b>
1011	<b>B</b>
1100	<b>C</b>
1101	<b>D</b>
1110	<b>E</b>
1111	<b>F</b>

$$\begin{aligned} 1A2F &= 1 \times 16^3 + A \times 16^2 + 2 \times 16^1 + F \times 16^0 \\ &= 1 \times 4096 + 10 \times 256 + 2 \times 16 + 15 \times 1 \\ &= 4096 + 2560 + 32 + 15 \\ &= 6703 \end{aligned}$$

# Decimal to Hex



Do:

Decimal  $\longrightarrow$  Binary

Binary  $\longrightarrow$  Hex

# Exercises



[http://www.science.smith.edu/dftwiki/index.php/  
CSC231\\_Review\\_of\\_hexadecimal\\_number\\_system](http://www.science.smith.edu/dftwiki/index.php/CSC231_Review_of_hexadecimal_number_system)



Convert these **hexadecimal** numbers to **binary**, and to **decimal**.

1111

1234

AAAA

F001

FFFF



Convert these **decimal** numbers to **binary**, and **hexadecimal**

65

127



What comes after these **hexadecimal** numbers, logically?

aaAA

9999

19F

1ABF

FFEF

F00F

ABCDEF



Perform the following additions in hex

$$\begin{array}{r} 1000 \\ + \text{AAAA} \\ \hline \end{array}$$

$$\begin{array}{r} 1234 \\ + \quad \text{F} \\ \hline \end{array}$$

$$\begin{array}{r} 1234 \\ + \text{ABCD} \\ \hline \end{array}$$

$$\begin{array}{r} \text{FFFF} \\ + \text{FFFF} \\ \hline \end{array}$$

We stopped here  
last time...



# More Arithmetic Instructions



# Intel Pentium 4 Northwood

## Buffer Allocation & Register Rename

Instruction Queue (for less critical fields of the uOps)

General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)

Floating Point, MMX, SSE2 Renamed Register File  
128 entries of 128 bit.

## uOp Schedulers

FP Move Scheduler:  
(8x8 dependency matrix)

Parallel (Matrix) Scheduler for the two double pumped ALU's

General Floating Point and Slow Integer Scheduler:  
(8x8 dependency matrix)

Load / Store uOp Scheduler:  
(8x8 dependency matrix)

Load / Store Linear Address Collision History Table

## Execution Pipeline Start

Register Alias History Tables (2x126)  
Register Alias Tables  
uOp Queue

## Instruction Trace Cache

Micro code Sequencer  
Micro code ROM & Flash  
Trace Cache Fill Buffers  
Distributed Tag comparators  
24 bit virtual Tags

## Trace Cache Access, next Address Predict

Trace Cache Branch Prediction Table (BTB), 512 entries.

Return Stacks (2x16 entries)

Trace Cache next IP's (2x)

Miscellaneous Tag Data

## Instruction Decoder

Up to 4 decoded uOps/cycle out, (from max. one x86 instr/cycle)  
Instructions with more than four are handled by Micro Sequencer

Trace Cache LRU bits

Raw Instruction Bytes in Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

## Instruction Fetch from L2 cache and Branch Prediction

Front End Branch Prediction Tables (BTB), shared, 4096 entries in total

Instruction TLB's 2x64 entry, fully associative for 4k and 4M pages. In: Virtual address [31:12]  
Out: Physical address [35:12] + 2 page level bits

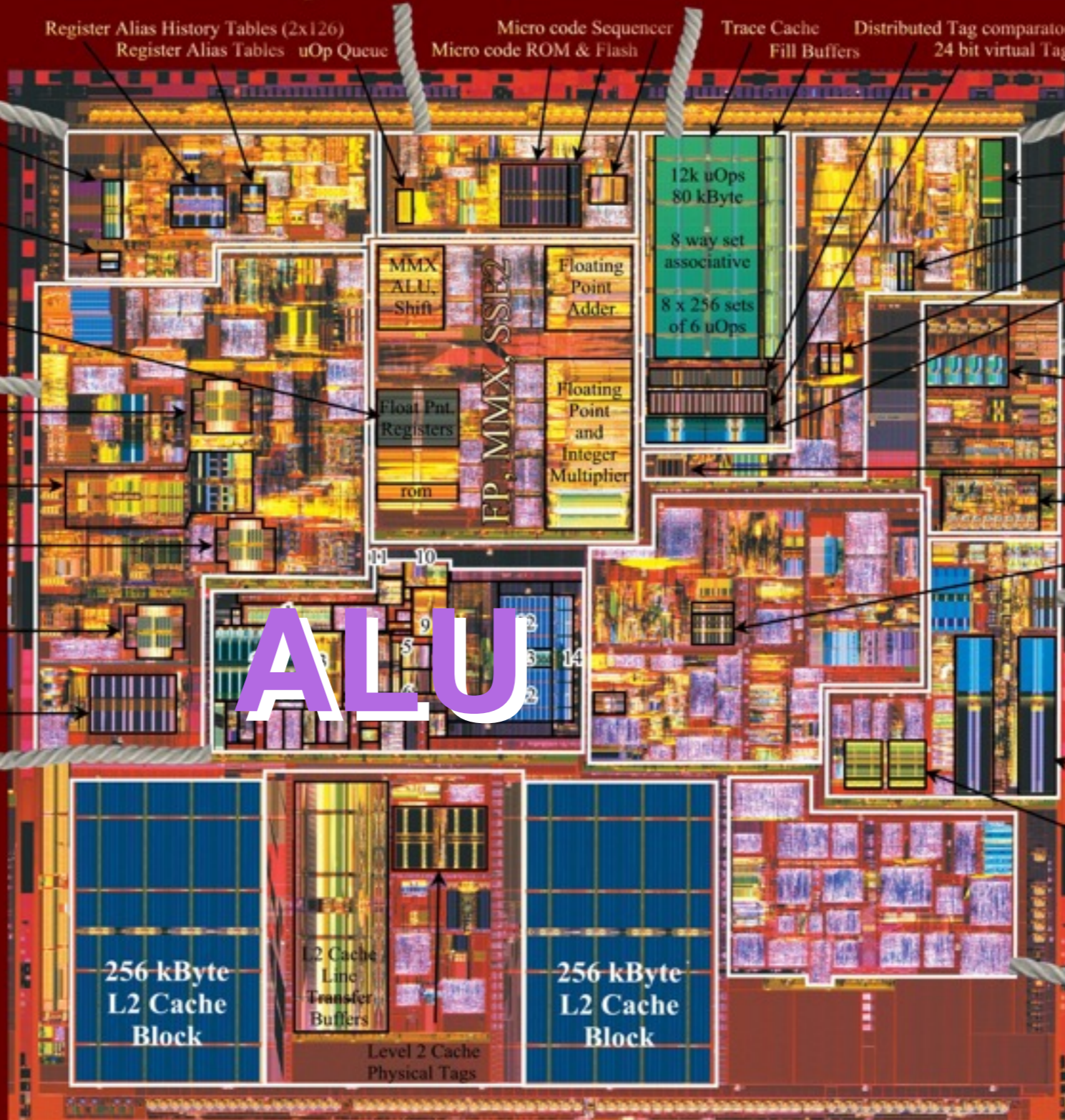
## Front Side Bus Interface, 400..800 MHz

## Integer Execution Core

- (1) uOp Dispatch unit & Replay Buffer  
Dispatches up to 6 uOps / cycle
- (2) Integer Renamed Register File  
128 entries of 32 bit + 6 status flags  
12 read ports and six write ports
- (3) Databus switch & Bypasses to and from the Integer Register File.
- (4) Flags, Write Back
- (5) Double Pumped ALU 0
- (6) Double Pumped ALU 1
- (7) Load Address Generator Unit
- (8) Store Address Generator Unit
- (9) Load Buffer ( 48 entries )
- (10) Store Buffer ( 24 entries )

- (11) ROB Reorder Buffer 3x42 entries
- (12) 8 kByte Level 1 Data cache

- (13) Summed Address Index decode and Way Predict
- (14) Cache Line Read / Write Transferbuffers and 256 bit wide bus to and from L2 cache



April 19, 2003 [www.chip-architect.com](http://www.chip-architect.com)



Right now,  
we are dealing only  
with **UNSIGNED** integers!

# inc

**inc operand**

```
inc reg8
inc reg16
inc reg32
inc mem8
inc mem16
inc mem32
```

```
alpha db 3
beta dw 4
x dd 0
```

```
inc al
inc cx
inc ebx
```

```
inc word[beta] ;beta <- 5
inc dword[x] ;x <- 1
```

# dec

## dec operand

```
dec    reg8
dec    reg16
dec    reg32
dec    mem8
dec    mem16
dec    mem32
```

```
alpha db    3
beta  dw    4
x     dd    6
```

```
dec    al      ; al <- al-1
dec    cx
dec    ebx
```

```
dec    word[beta] ; beta <- 3
dec    dword[x]   ; x <- 5
```

mul

**mul operand**

```
mul    reg8  
mul    reg16  
mul    reg32  
mul    mem8  
mul    mem16  
mul    mem32
```

# Observation

$$\begin{array}{r} 1001 \\ \times 1110 \\ \hline \end{array}$$

# mul

## mul operand

```
mul    reg8
mul    reg16
mul    reg32
mul    mem8
mul    mem16
mul    mem32
```

$edx:eax \leftarrow operand_{32} * eax$   
 $dx:ax \leftarrow operand_{16} * ax$   
 $ax \leftarrow operand_8 * al$

alpha	db	3
beta	dw	4
x	dd	6

```
mul    byte[alpha]    ;ax <- al*alpha
mul    ebx             ;edx:eax <-
                        ;      ebx*eax
```



This has tremendously  
important **consequences!**



```
public class JavaLimits {  
  
    public static void main(String[] args) {  
        // -----  
        // a multiplication of ints  
        int x = 0x30000001;  
        int y = 0x30000001;  
  
        System.out.println( "x = " + x );  
        System.out.println( "y = " + y );  
  
        int z = x * y;  
  
        System.out.println( "z = " + z );  
        System.out.println();  
    }  
}
```

```

public class JavaLimits {

    public static void main(String[] args) {
        // -----
        // a multiplication of ints
        int x = 0x30000001;
        int y = 0x30000001;

        System.out.println( "x = " + x );
        System.out.println( "y = " + y );

        int z = x * y;

        System.out.println( "z = " + z );
        System.out.println();
    }
}

```

```

x = 805306369
y = 805306369
z = 1610612737

```

How big is a 32-bit  
**int**?

# Ranges (Unsigned Integers)

8 bits	0 - 255
16 bits	0 - 65,535
32 bits	0 - 4,294,967,295

# div

div	reg8
div	reg16
div	reg32
div	mem8
div	mem16
div	mem32

## div operand

R : Q

edx:eax  $\leftarrow$  edx:eax / operand<sub>32</sub>

dx:ax  $\leftarrow$  dx:ax / operand<sub>16</sub>

ah:al  $\leftarrow$  ax / operand<sub>8</sub>

```
alpha db 3
beta dw 4
x dd 6
;compute beta/alpha
    mov ax, word[beta]
    div byte[alpha]
;quotient in al
;remainder in ah
```

# Exercise

Compute  $x = 2 * \text{alpha} + 3 * \text{beta} + x - 1$

alpha	db	3
beta	dw	4
x	dd	6



# Logical Instructions

**AND, OR, NOT, XOR**

10010  
and 11100  
          

10010  
or 11100  
          

10010  
xor 11100  
          

not 11100



# and

**and dest, src**

```
and    op8,op8  
and    op16,op16  
and    op32,op32  
  
op: mem, reg, imm
```

```
alpha db 3  
beta  dw 4  
x     dd 6
```

```
and    byte[alpha], 4  
mov    ax, 0x1234  
and    ax, 0xFF00  
  
and    dword[x], 1
```

or

**or dest, src**

```
or      op8,op8
or      op16,op16
or      op32,op32

op: mem, reg, imm
```

```
alpha db 3
beta  dw 4
x     dd 0xF06
```

```
or    byte[alpha], 4
mov  ax, 0x1234
or    ax, 0xFF00

or    dword[x], 15
```

# xor

**xor dest, src**

```
xor    op8,op8
xor    op16,op16
xor    op32,op32

op: mem, reg, imm
```

```
alpha db 3
beta  dw 4
x     dd 0xF06
```

```
xor    byte[alpha], 4
mov    ax, 0x1234
xor    ax, 0xFF00

xor    dword[x], 15
```

not

**not oprnd**

```
not    op8
not    op16
not    op32

op: mem, reg
```

```
alpha db 3
beta  dw 4
x     dd 0xF06
```

```
not    byte[alpha]
mov    ax, 0x1234
not    ax
```

```
not    dword[x]
```

Instruction	Feature
<b>AND</b>	Good for setting bits to 0
<b>OR</b>	Good for setting bits to 1
<b>XOR</b>	Good for flipping bits
<b>NOT</b>	Good for complementing all the bits



# NEGATIVE NUMBERS



0100 1100

# Signed Magnitude

0000

0001

0010

0011

0100

0101

0110

0111

1000

1001

1010

1011

1100

1101

1110

1111



# Signed Magnitude

0	000
0	001
0	010
0	011
0	100
0	101
0	110
0	111
1	000
1	001
1	010
1	011
1	100
1	101
1	110
1	111

# Signed Magnitude

0 000  
0 001  
0 010  
0 011  
0 100  
0 101  
0 110  
0 111  
1 000  
1 001  
1 010  
1 011  
1 100  
1 101  
1 110  
1 111

$$\begin{array}{r} 0\ 010 \\ +\ 0\ 011 \\ \hline \end{array}$$

$$\begin{array}{r} 0\ 110 \\ +\ 1\ 011 \\ \hline \end{array}$$

# 1's Complement

0	000
0	001
0	010
0	011
0	100
0	101
0	110
0	111
1	000
1	001
1	010
1	011
1	100
1	101
1	110
1	111

# 1's Complement

0 000  
0 001  
0 010  
0 011  
0 100  
0 101  
0 110  
0 111  
1 000  
1 001  
1 010  
1 011  
1 100  
1 101  
1 110  
1 111

$$\begin{array}{r} 0\ 010 \\ +\ 0\ 011 \\ \hline \end{array}$$

$$\begin{array}{r} 0\ 110 \\ +\ 1\ 011 \\ \hline \end{array}$$

$$\begin{array}{r} 0\ 110 \\ +\ 1\ 111 \\ \hline \end{array}$$

# 2's Complement

0	000
0	001
0	010
0	011
0	100
0	101
0	110
0	111
1	000
1	001
1	010
1	011
1	100
1	101
1	110
1	111

# 2's Complement

0 000  
0 001  
0 010  
0 011  
0 100  
0 101  
0 110  
0 111  
1 000  
1 001  
1 010  
1 011  
1 100  
1 101  
1 110  
1 111

$$\begin{array}{r} 0\ 010 \\ +\ 0\ 011 \\ \hline \end{array}$$

$$\begin{array}{r} 0\ 110 \\ +\ 1\ 011 \\ \hline \end{array}$$

$$\begin{array}{r} 0\ 110 \\ +\ 1\ 111 \\ \hline \end{array}$$

# Range of 2's Complement Number Systems