IA32 Assembly Language Programming
Part 1

198:231 Introduction to Computer Organization
Lecture 4

Instructor:
Nicole Hynes
nicole.hynes@rutgers.edu
Learning What’s Under the Hood

Will discuss this next

Language Levels
- High-Level Language
- Assembly Language
- Machine Language

Service Levels
- Application Program
- Operating System
- Hardware

Under the hood
Language Levels

High-Level Language

C program

```c
#include <stdio.h>
int a, b, c;
int main () {
    a = 15;
    b = a << 2;
    c = a * b;
    return 0;
}
```

Compiler

Assembly Language

Entrypoint: main

```
.text
.globl main
main:
    pushl %ebp
    movl %esp, %ebp
    movl $15, a
    sall $2, %eax
    movl %eax, b
    movl a, %edx
    movl b, %eax
    imull %edx, %eax
    movl %eax, c
    movl $0, %eax
    popl %ebp
    ret
```

Executable object program

```
0: 55
1: 89 e5
3: c7 05 00 00 00 00 0f a: 00 00 00
d: a1 00 00 00 00
12: c1 e0 02
15: a3 00 00 00 00
1a: 8b 15 00 00 00 00
20: a1 00 00 00 00
25: 0f af c2
28: a3 00 00 00 00
2d: b8 00 00 00 00
32: 5d
33: c3
```
Assembly Language

- Low-level symbolic programming language
- Each “statement” corresponds to a machine instruction of the target instruction set architecture (ISA)

Architecture-specific:

- IA32 (aka Intel x86-32)
  - Supported in Intel Pentium 4 and older microprocessors
- AMD64
  - Advanced Micro Devices’ 64-bit extension to IA32
- Intel 64 (aka Intel x86-64)
  - Intel’s version of AMD64; implemented in Xeon, Core, and newer generations
  - Maintains backward compatibility with IA32
- Many others: Motorola 68k, IBM PowerPC, Sun SPARC, DEC Alpha, Acorn ARM
Instruction Set Architectures

- Instruction set architecture (ISA)
  - Defines the set of instructions that can be executed by the target machine
  - Includes native data types, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O

- CISC vs. RISC
  - CISC = Complex Instruction Set Computers
    - Evolved earlier
  - RISC = Reduced Instruction Set Computers
    - Developed in the 1980s
# Instruction Set Architectures

<table>
<thead>
<tr>
<th>CISC</th>
<th>Early RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large number of instructions</td>
<td>Fewer instructions</td>
</tr>
<tr>
<td>Variable length instruction format</td>
<td>Fixed length instruction format</td>
</tr>
<tr>
<td>Many complex addressing modes</td>
<td>Fewer and simpler addressing modes</td>
</tr>
<tr>
<td>Instructions can operate directly on</td>
<td>Only load/store instructions can access memory; rest operate only on</td>
</tr>
<tr>
<td>register and memory operands</td>
<td>registers (load/store architecture)</td>
</tr>
<tr>
<td>Stack- (i.e., memory-) intensive procedure</td>
<td>Register-intensive procedure linkage</td>
</tr>
<tr>
<td>linkage</td>
<td></td>
</tr>
<tr>
<td>Instructions execute in multiple clock</td>
<td>Instructions execute in single clock cycles</td>
</tr>
<tr>
<td>cycles</td>
<td></td>
</tr>
<tr>
<td>Attempts to shorten program execution</td>
<td>Attempts to shorten program execution time by reducing the clock cycles</td>
</tr>
<tr>
<td>time by reducing the number of instructions in the program</td>
<td>per instructions</td>
</tr>
<tr>
<td>Examples: IBM System/360, PDP-11,</td>
<td>Examples: MIPS, PowerPC, SPARC, Alpha, PA-RISC, ARM</td>
</tr>
<tr>
<td>Motorola 68k, <strong>Intel IA32</strong></td>
<td></td>
</tr>
</tbody>
</table>
IA32 and Intel 64

- Intel 64 adopts many (but not all) of the features of early RISC machines.
- Will study both IA32 and Intel 64 instruction set architectures.
- Will use IA32 as vehicle to learn the processes of compilation, assembly, linking, loading, and program execution.
- Will describe salient differences between IA32 and Intel 64 and also briefly discuss Intel 64 assembly language programming.
- Will study IA32 first.
Assembly Programmer’s View

Programmer-Visible State

- **PC:** Program counter
  - Address of next instruction
  - Called “EIP” (IA32) or “RIP” (x86-64)

- **Register file**
  - Heavily used program data

- **Condition codes**
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

**Memory**

- Byte addressable array
- Code, user data, (some) OS data
- Includes stack used to support procedures
IA32 Instruction Set Architecture

- **Types of instructions**
  - Data movement
  - Arithmetic and logical
  - Control
    - Conditional branches
    - Unconditional jumps to/from procedures

- **Variable length encoding of instructions: 1 - 15 bytes**

- **Minimal data types**
  - "Integer" data of 1, 2 or 4 bytes
    - Data values
    - Addresses (pointers)
  - Floating point data of 4, 8 or 10 bytes
  - No aggregate types such as arrays or structures
    - Just contiguously allocated bytes of memory

- **Data stored in registers or memory**

- **Will first discuss integer instructions.**
IA32 Integer Registers

- **%eax**
- **%ecx**
- **%edx**
- **%ebx**
- **%esi**
- **%edi**
- **%esp**
- **%ebp**

**Origin (mostly obsolete)**
- accumulate
- counter
- data
- base
- source index
- destination index
- stack pointer
- base pointer

**16-bit virtual registers (backwards compatibility)**
- %ax
- %ch
- %dh
- %bl
- %ah
- %cl
- %dl
- %al
Move Instruction

**movx src, dest**

- **Copy src to dest**
- **Operand sizes**
  - `movb` move byte
  - `movw` move word – 2 bytes
  - `movl` move long (aka double word) – 4 bytes
- **Operand types**
  - *Immediate*: constant integer data
    - Example: `movl $-125, %eax`
    - Integer constant prefixed with `$`'
  - *Register*: one of 8 integer registers
    - Example: `movl %eax, %edx`
    - But `%esp` and `%ebp` reserved for special use
  - *Memory*: contents of memory at address given by register
    - Example: `movl (%eax), %ecx`
    - Move to `%ecx` the 4-byte contents of memory address stored `%eax`
Move Operand Combinations

\[
\begin{array}{cccc}
\text{Source} & \text{Dest} & \text{Src, Dest} & \text{C Analog} \\
\text{movl} & \text{Reg} & \text{movl } \$0x4,\%eax & \text{temp } = \text{ } 0x4; \\
& \text{Imm} & \text{movl } \$-147,\(\%eax) & \text{*p } = \text{ } -147; \\
& \text{Mem} & \text{movl } \%eax,\%edx & \text{*p } = \text{ } \text{temp}; \\
\text{Reg} & \text{Reg} & \text{movl } \%eax,\%edx & \text{temp } = \text{ } \text{temp}; \\
& \text{Mem} & \text{movl } \%eax,\(\%edx) & \text{temp } = \text{ } \text{*p}; \\
\text{Mem} & \text{Reg} & \text{movl } \(\%eax),\%edx & \text{temp } = \text{ } \text{*p}; \\
\end{array}
\]

Cannot do memory-memory transfer with a single instruction
### Immediate Operands

- **Immediate operand = constant**
  - `movl $125,%eax`
    - $125_{10} = 0000007D_{16}$
  - `movl $-125,%eax`
    - $-125_{10} = FFFFFFF83_{16}$
  - Cannot be destination
    - `movl %eax,$125` wrong!

- **Expressing integer constants**
  - Decimal: $125$ $-125$
  - Hexadecimal: prefix with 0x or 0X (“zero-x”) $0x7D$ $-0x7D$
  - Binary: prefix with 0b or 0B (“zero-b”) $0b1111101$ $-0b1111101$
  - Octal: leading zero followed by octal digits $0175$ $-0175$

- **Expressing character constants**
  - `movb $'Z'$,%al`
    - $'Z'$ = 0x5A

---

\[ \begin{array}{c}
\text{movl } \$125,\%eax \\
125_{10} = 0000007D_{16} \\
\text{movl } \$-125,\%eax \\
-125_{10} = FFFFFFF83_{16} \\
\text{movl } \%eax,\$125 \text{ wrong!}
\end{array} \]
Register Operands

- Size specification must match register
  - `movb` for 1-byte registers only (e.g. `%ah, %dl`)
  - `movw` for 2-byte registers only (e.g. `%cx`)
  - `movl` for 4-byte registers only (e.g. `%ebx`)

- Examples
  - `movb $24, %eax` wrong!
  - `movb $24, %al`
  - `movl $24, %eax`
  - `movw %cx, %edx` wrong!
  - `movw %cx, %dx`
  - `movl %eax, %ebx`
Register Operands

- Can move data from a smaller register to a larger register using special move instructions:
  - **movsx src,dest**
    - Copy and sign-extend src to dest
      - `movsbw` move sign-extended byte to word
      - `movsbl` move sign-extended byte to long/double word
      - `movswl` move sign-extended word to long/double word
  - **movzx src,dest**
    - Copy and zero-extend src to dest
      - `movzbw` move zero-extended byte to word
      - `movzbl` move zero-extended byte to long/double word
      - `movzwl` move zero-extended word to long/double word
Register Operands

Example

- Assume that before each instruction below is executed, \%eax and \%ecx have the following values:

\[ \begin{array}{c}
\%eax & \%ecx \\
EE5599AA & 11223344 \\
\text{before} & \text{before}
\end{array} \]

- `movb %ah,%cl`

\[ \begin{array}{c}
\%eax & \%ecx \\
EE5599AA & 11223399 \\
\text{after} & \text{after}
\end{array} \]

- `movsbw %al,%cx`

\[ \begin{array}{c}
\%eax & \%ecx \\
EE5599AA & 1122FFAA \\
\text{after} & \text{after}
\end{array} \]

- `movzwl %ax, %ecx`

\[ \begin{array}{c}
\%eax & \%ecx \\
EE5599AA & 000099AA \\
\text{after} & \text{after}
\end{array} \]
A Note on IA32 Assembly Language Syntax

- There are two widely adopted conventions for writing IA32 assembly language instructions:
  - GNU or AT&T syntax – used by the GNU C compiler (gcc) and the GNU Assembler (gas); this is the syntax used in this course
  - Intel syntax – used by Microsoft’s Macro Assembler (masm); most Intel documents use the Intel syntax, including the Intel IA32 manuals

- Main difference between GNU/AT&T syntax and Intel syntax is that the order of the source and destination operands are reversed; e.g.,

<table>
<thead>
<tr>
<th>GNU/AT&amp;T Syntax</th>
<th>Intel Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $24, %eax</td>
<td>mov eax, 24</td>
</tr>
</tbody>
</table>
Programmer’s View of IA32 Memory

- **To the programmer:**
  - The IA32 consists of $2^{32}$ bytes = 4 GB (gigabytes) of memory, which can be viewed as a linear array.
  - Memory is *byte-addressable*: each byte has a unique *address* (think “array index”), from 0x00000000 to 0xFFFFFFFF.
  - This memory abstraction is called *virtual memory* and the corresponding set of addresses is called the *virtual address space*.

- **In reality:**
  - Structure and size of *physical memory* depends on the actual machine and the operating system.
  - Physical memory is shared by multiple programs (processes) running at the same time.
  - The operating system manages on-the-fly the sharing and allocation of physical memory amongst multiple processes.
  - Operating systems course will discuss this topic in depth.
Programmer’s View of IA32 Memory

Will depict memory as a linear array:

```
0xFFFFFFFF
0xFFFFFFFE
...  
0x3456442B  AB
0x3456442A  CD
0x34564429  99
0x34564428  88
...  
0x00000003  66
0x00000002  55
0x00000001  44
0x00000000  33
0xFFFFFFFC
0xFFFFFFF8
...  
0x0000000C
0x00000008
0x00000004
0x00000000
```

or

```
0xFFFFFFFF
0xFFFFFFFE
...  
0x34564434
0x34564430
0x3456442C
0x34564428
...  
0x0000000C
0x00000008
0x00000004
0x00000000
```
Memory Operands

- IA32 instructions allow operands that reside in memory.

- A memory operand has two attributes:
  - 32-bit effective address (or simply address)
  - size (byte, word, or long/double word)

- Address of memory operand is specified using one of several addressing modes.

- Size of memory operand is determined by instruction, e.g.:
  - `movb addr, %al` move to %al the contents of $M[addr]$
  - `movw addr, %ax` move to %ax the contents of $M[addr:addr+1]$
  - `movl addr, %eax` move to %eax the contents of $M[addr:addr+3]$

- **Notation**: $M[a:b]$ means the contents of the contiguous sequence of bytes from memory address $a$ to memory address $b$. 
# IA32 Addressing Modes

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Effective Address (EA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>$Imm$</td>
<td>$EA = Imm$</td>
</tr>
<tr>
<td>Indirect</td>
<td>($E_a$)</td>
<td>$EA = R[E_a]$</td>
</tr>
<tr>
<td>Base + Displacement</td>
<td>$Imm(E_b)$</td>
<td>$EA = Imm + R[E_b]$</td>
</tr>
<tr>
<td>Indexed</td>
<td>($E_b, E_i$)</td>
<td>$EA = R[E_b] + R[E_i]$</td>
</tr>
<tr>
<td>Indexed</td>
<td>$Imm(E_b, E_i)$</td>
<td>$EA = Imm + R[E_b] + R[E_i]$</td>
</tr>
<tr>
<td>Scaled Indexed</td>
<td>($E_i, s$)</td>
<td>$EA = R[E_i] \times s$</td>
</tr>
<tr>
<td>Scaled Indexed</td>
<td>$Imm(E_i, s)$</td>
<td>$EA = Imm + (R[E_i] \times s)$</td>
</tr>
<tr>
<td>Scaled Indexed</td>
<td>($E_b, E_i, s$)</td>
<td>$EA = R[E_b] + (R[E_i] \times s)$</td>
</tr>
<tr>
<td>Scaled Indexed</td>
<td>$Imm(E_b, E_i, s)$</td>
<td>$EA = Imm + R[E_b] + (R[E_i] \times s)$</td>
</tr>
</tbody>
</table>

**Legend:**
- $Imm$ = immediate integer constant; at most 32 bits
- $E_a$ = 32-bit register; $R[E_a]$ = contents of register $E_a$
- $s$ = scaling factor; must be 1, 2, 4, or 8
Absolute Addressing

Example 1: movb \texttt{0x34564428},\%al

- Effective address EA = \texttt{0x34564428}
- Note: no ‘$’ before immediate constant

```
Example 1: movb 0x34564428, %al

- Effective address EA = 0x34564428
- Note: no ‘$’ before immediate constant

Increasing addresses

Memory

\begin{tabular}{|c|}
\hline
0x3456442B & AB \\
0x3456442A & CD \\
0x34564429 & 99 \\
0x34564428 & 88 \\
\hline
\end{tabular}

\%eax | 11 | 22 | 33 | 44 | before
\%al

\%eax | 11 | 22 | 33 | 88 | after
\%al
```
Absolute Addressing

Example 2: `movw 0x34564428, %ax`

IA32 byte ordering is **Little Endian**: least significant byte is at the lowest address.

(Other machines are Big Endian: most significant byte is at the lowest address. E.g., IBM PowerPC, Sun SPARC, MIPS.)
Example 3: `movl 0x34564428,%eax`

Memory:

```
  .  .  .
0x3456442B AB
0x3456442A CD
0x34564429 99
0x34564428 88
  .  .  .
```

%eax before:

```
  11 22 33 44
```

%eax after:

```
  AB CD 99 88
```

Note again **Little Endian** byte ordering: **least significant byte** is at the **lowest address**.
Indirect Addressing

Example: `movl (%ebx),%eax`
- EA = R[ebx] (i.e., the 32-bit contents of ebx)
- In other words, ebx is a pointer to the memory operand.

Memory

%ebx 34 56 44 28

0x3456442B
0x3456442A
0x34564429
0x34564428

AB
CD
99
88

%eax 11 22 33 44 before

%eax AB CD 99 88 after
More general form of indirect addressing

Example: `movl 8(%ebx), %eax`
- EA = R[%ebx] + 8
- `%ebx` is the base register; 8 is the displacement
- Base register value doesn’t change; displ. at most 32 bits
Indexed Addressing

Example: `movl 8(%ebx,%ecx),%eax`

- EA = R[%ebx] + R[%ecx] + 8
- %ebx = base register; %ecx = index register; 8 = displacement
- Values of base and index registers do not change

```
%ebx  34 56 44 00
+     00 00 00 20
%ecx  00 00 00 20
```

```
Memory

| 0x3456442B | AB |
| 0x3456442A | CD |
| 0x34564429 | 99 |
| 0x34564428 | 88 |

%eax  11 22 33 44  before

%eax  AB CD 99 88  after
```
## Scaled Indexed Addressing

**Example:** `movl 8(,%ebx,%ecx,4),%eax`

- **EA = R[%ebx] + (R[%ecx]×4) + 8**
- `%ebx` = base register; `%ecx` = index register; 8 = displacement
- 4 = scale (can only be 1, 2, 4, or 8)

![Diagram showing scaled indexed addressing](image)
IA32 Addressing Modes

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Effective Address (EA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>$Imm$</td>
<td>$EA = Imm$</td>
</tr>
<tr>
<td>Indirect</td>
<td>$(E_a)$</td>
<td>$EA = R[E_a]$</td>
</tr>
<tr>
<td>Base + Displacement</td>
<td>$Imm(E_b)$</td>
<td>$EA = Imm + R[E_b]$</td>
</tr>
<tr>
<td>Indexed</td>
<td>$(E_b, E_i)$</td>
<td>$EA = R[E_b] + R[E_i]$</td>
</tr>
<tr>
<td>Indexed</td>
<td>$Imm(E_b, E_i)$</td>
<td>$EA = Imm + R[E_b] + R[E_i]$</td>
</tr>
<tr>
<td>Scaled Indexed</td>
<td>$(, E_i, s)$</td>
<td>$EA = R[E_i] \times s$</td>
</tr>
<tr>
<td>Scaled Indexed</td>
<td>$Imm(, E_i, s)$</td>
<td>$EA = Imm + (R[E_i] \times s)$</td>
</tr>
<tr>
<td>Scaled Indexed</td>
<td>$(E_b, E_i, s)$</td>
<td>$EA = R[E_b] + (R[E_i] \times s)$</td>
</tr>
<tr>
<td>Scaled Indexed</td>
<td>$Imm(E_b, E_i, s)$</td>
<td>$EA = Imm + R[E_b] + (R[E_i] \times s)$</td>
</tr>
</tbody>
</table>

Legend:
- $Imm$ = immediate integer constant; at most 32 bits
- $E_a$ = 32-bit register; $R[E_a]$ = contents of register $E_a$
- $s$ = scaling factor; must be 1, 2, 4, or 8
## Load Effective Address Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>leal src,dest</td>
<td>Load effective address</td>
<td>dest ← address(src)</td>
</tr>
</tbody>
</table>

- Computes the effective address given by the source and stores it in the destination; source must specify a memory address.
- Destination must be a 32-bit register.
- **Example:**
  - Suppose `%ebx` contains the value `0x34564428` and memory has the contents shown:
    - `movl (%ebx), %eax`:
      - Result in `%eax`: `0xABCD9988`
      - Memory: `0x34564428` (88), `0x34564429` (99), `0x3456442A` (AB), `0x3456442B` (CD) in `%eax`
    - `leal (%ebx), %eax`:
      - Result in `%eax`: `0x34564428`
      - Memory: `0x34564428` (88), `0x34564429` (99), `0x3456442A` (AB), `0x3456442B` (CD) in `%eax`

- In other words, `leal` does **not** access memory but only copies the address specified by the source to the destination.
# Arithmetic & Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>incx</code> dest</td>
<td>Increment</td>
<td>dest ← dest + 1</td>
</tr>
<tr>
<td><code>decx</code> dest</td>
<td>Decrement</td>
<td>dest ← dest − 1</td>
</tr>
<tr>
<td><code>negx</code> dest</td>
<td>Negate</td>
<td>dest ← −dest</td>
</tr>
<tr>
<td><code>addx</code> src,dest</td>
<td>Add</td>
<td>dest ← dest + src</td>
</tr>
<tr>
<td><code>subx</code> src,dest</td>
<td>Subtract</td>
<td>dest ← dest − src</td>
</tr>
<tr>
<td><code>imulx</code> src,dest</td>
<td>Multiply</td>
<td>dest ← dest × src</td>
</tr>
<tr>
<td><code>notx</code> dest</td>
<td>Not</td>
<td>dest ← ~dest</td>
</tr>
<tr>
<td><code>andx</code> src,dest</td>
<td>And</td>
<td>dest ← dest &amp; src</td>
</tr>
<tr>
<td><code>orx</code> src,dest</td>
<td>Or</td>
<td>dest ← dest</td>
</tr>
<tr>
<td><code>xorx</code> src,dest</td>
<td>Exclusive or</td>
<td>dest ← dest ^ src</td>
</tr>
</tbody>
</table>

**Notes:**
- `x` = b (byte), w (word), or l (long or double word)
- `src` and `dest` cannot be both memory operands
- No two-operand divide instruction
Arithmetic & Logical Instructions

- Arithmetic instructions – self-explanatory; operate on signed integers in two’s-complement form
- Logical instructions – apply bitwise the following boolean operations:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>~a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Not

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- Or

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- And

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a &amp; b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Xor

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a ^ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
## Arithmetic & Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>%eax before</th>
<th>%eax after</th>
</tr>
</thead>
<tbody>
<tr>
<td>incl %eax</td>
<td>0x11223344</td>
<td>0x11223345</td>
</tr>
<tr>
<td>decl %eax</td>
<td>0x11223344</td>
<td>0x11223343</td>
</tr>
<tr>
<td>negl %eax</td>
<td>0x11223344</td>
<td>0xeeddccbC</td>
</tr>
<tr>
<td>addl $4,%eax</td>
<td>0x11223344</td>
<td>0x11223348</td>
</tr>
<tr>
<td>subl $4,%eax</td>
<td>0x11223344</td>
<td>0x11223340</td>
</tr>
<tr>
<td>imull $4,%eax</td>
<td>0x11223344</td>
<td>0x4488cd10</td>
</tr>
<tr>
<td>notl %eax</td>
<td>0x11223344</td>
<td>0xeeddccbB</td>
</tr>
<tr>
<td>andl $0xffff00ff00,%eax</td>
<td>0x11223344</td>
<td>0x11003300</td>
</tr>
<tr>
<td>orl $0xffff00ff00,%eax</td>
<td>0x11223344</td>
<td>0xffff22ff44</td>
</tr>
<tr>
<td>xorl $0xffff00ff00,%eax</td>
<td>0x11223344</td>
<td>0xee22cc44</td>
</tr>
</tbody>
</table>
Special Arithmetic Instructions: One-Operand Multiply Instruction

- \texttt{mulx src} multiplies \texttt{src} by the corresponding part of the \texttt{%eax} register (as determined by size specifier \texttt{x})
- \texttt{src} – register or memory operand
- Result is twice the size and stored in \texttt{%edx} and \texttt{%eax}
- \texttt{mulx} performs unsigned multiplication
- \texttt{imulx} performs signed multiplication

<table>
<thead>
<tr>
<th>Instruction</th>
<th>mulb</th>
<th>mulw</th>
<th>mull</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other operand:</td>
<td>\texttt{%al}</td>
<td>\texttt{%ax}</td>
<td>\texttt{%eax}</td>
</tr>
<tr>
<td>Higher part of result stored in:</td>
<td>\texttt{%ah}</td>
<td>\texttt{%dx}</td>
<td>\texttt{%edx}</td>
</tr>
<tr>
<td>Lower part of result stored in:</td>
<td>\texttt{%al}</td>
<td>\texttt{%ax}</td>
<td>\texttt{%eax}</td>
</tr>
</tbody>
</table>
Special Arithmetic Instructions
One-Operand Multiply Instruction

In the following assume %ecx contains value 4 (decimal).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>%edx before</th>
<th>%eax before</th>
<th>%edx after</th>
<th>%eax after</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulb %cl</td>
<td>0x00000000</td>
<td>0xccccccc</td>
<td>0x00000000</td>
<td>0xcccc0330</td>
</tr>
<tr>
<td></td>
<td>(0)</td>
<td>(204)</td>
<td>(0)</td>
<td>(816)</td>
</tr>
<tr>
<td>mulw %cx</td>
<td>0x00000000</td>
<td>0xccccccc</td>
<td>0x00000003</td>
<td>0xcccc3330</td>
</tr>
<tr>
<td></td>
<td>(0)</td>
<td>(54,428)</td>
<td>(209,712)</td>
<td></td>
</tr>
<tr>
<td>mull %ecx</td>
<td>0x00000000</td>
<td>0xccccccc</td>
<td>0x00000003</td>
<td>0x33333330</td>
</tr>
<tr>
<td></td>
<td>(0)</td>
<td>(3,435,973,836)</td>
<td>(13,743,895,344)</td>
<td></td>
</tr>
<tr>
<td>imulb %ecx</td>
<td>0x00000000</td>
<td>0xccccccc</td>
<td>0x00000000</td>
<td>0xccccff30</td>
</tr>
<tr>
<td></td>
<td>(0)</td>
<td>(-52)</td>
<td>(0)</td>
<td>(-208)</td>
</tr>
<tr>
<td>imulw %cx</td>
<td>0x00000000</td>
<td>0xccccccc</td>
<td>0x0000ffff</td>
<td>0xcccc3330</td>
</tr>
<tr>
<td></td>
<td>(0)</td>
<td>(-13,108)</td>
<td>(-52,432)</td>
<td></td>
</tr>
<tr>
<td>imull %ecx</td>
<td>0x00000000</td>
<td>0xccccccc</td>
<td>0xffffffff</td>
<td>0x33333330</td>
</tr>
<tr>
<td></td>
<td>(0)</td>
<td>(-858,993,460)</td>
<td>(-3,435,973,840)</td>
<td></td>
</tr>
</tbody>
</table>
Special Arithmetic Instructions: One-Operand Divide Instruction

- Performs integer division with two results: quotient and remainder
- Divisor is src, which may be a register or a memory operand
- Dividend, quotient, and remainder are (implicitly) stored in the following registers, as determined by size specifier x
- \texttt{divx} performs unsigned division
- \texttt{idivx} performs signed division

<table>
<thead>
<tr>
<th>Instruction</th>
<th>\texttt{divb}</th>
<th>\texttt{divw}</th>
<th>\texttt{divl}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dividend in:</td>
<td>%ax</td>
<td>%dx:%ax</td>
<td>%edx:%eax</td>
</tr>
<tr>
<td>Remainder in:</td>
<td>%ah</td>
<td>%dx</td>
<td>%edx</td>
</tr>
<tr>
<td>Quotient in:</td>
<td>%al</td>
<td>%ax</td>
<td>%eax</td>
</tr>
</tbody>
</table>

Note: The colon (:) means concatenation.
Special Arithmetic Instructions: One-Operand Divide Instruction

In the following assume `%ecx` contains value 4 (decimal).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>%edx</th>
<th>%eax</th>
<th>%edx</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>before</td>
<td>before</td>
<td>after</td>
<td>after</td>
</tr>
<tr>
<td><strong>divb %cl</strong></td>
<td>0x00000000</td>
<td>0x00000333</td>
<td>0x00000000</td>
<td>0x000003cc</td>
</tr>
<tr>
<td></td>
<td>(0)</td>
<td>(D = 819)</td>
<td>(0)</td>
<td>(R = 3, Q = 204)</td>
</tr>
<tr>
<td><strong>divw %cx</strong></td>
<td>0x00000003</td>
<td>0x00000333</td>
<td>0x00000003</td>
<td>0x0000cccc</td>
</tr>
<tr>
<td></td>
<td>(D = 209,715)</td>
<td>(R = 3)</td>
<td>(Q = 54,428)</td>
<td></td>
</tr>
<tr>
<td><strong>divl %ecx</strong></td>
<td>0x00000003</td>
<td>0x33333333</td>
<td>0x00000003</td>
<td>0xcccccccc</td>
</tr>
<tr>
<td></td>
<td>(D = 13,743,895,347)</td>
<td>(R = 3)</td>
<td>(Q = 3,435,973,836)</td>
<td></td>
</tr>
<tr>
<td><strong>idivb %cl</strong></td>
<td>0x00000000</td>
<td>0x0000ff2d</td>
<td>0x00000000</td>
<td>0x0000fdcc</td>
</tr>
<tr>
<td></td>
<td>(0)</td>
<td>(D = -211)</td>
<td>(0)</td>
<td>(R = -3, Q = -52)</td>
</tr>
<tr>
<td><strong>idivw %cx</strong></td>
<td>0xffff</td>
<td>0x0000332d</td>
<td>0xffff</td>
<td>0x0000cccc</td>
</tr>
<tr>
<td></td>
<td>(D = -52,435)</td>
<td>(R = -3)</td>
<td>(Q = -13,108)</td>
<td></td>
</tr>
<tr>
<td><strong>idivl %ecx</strong></td>
<td>0xffffffff</td>
<td>0x3333332d</td>
<td>0xffffffff</td>
<td>0xcffffff</td>
</tr>
<tr>
<td></td>
<td>(D = -3,435,973,843)</td>
<td>(R = -3)</td>
<td>(Q = -858,993,460)</td>
<td></td>
</tr>
</tbody>
</table>

Note: A **Divide Error Exception (#DE)** is raised if the divisor is 0 or the result cannot be represented in specified number of bits.
Special Arithmetic Instructions: Convert Instruction

- Useful for signed divide instruction
- Sign-extends integer stored in 1, 2, or 4 bytes of `%eax` register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cbtw</code></td>
<td><code>R[%ax] ← Sign-Extend(R[%al])</code></td>
</tr>
<tr>
<td><code>cwtx</code></td>
<td><code>R[%eax] ← Sign-Extend(R[%ax])</code></td>
</tr>
<tr>
<td><code>cwtd</code></td>
<td><code>R[%dx]:R[%ax] ← Sign-Extend(R[%ax])</code></td>
</tr>
<tr>
<td><code>cltd</code></td>
<td><code>R[%edx]:R[%eax] ← Sign-Extend(R[%eax])</code></td>
</tr>
</tbody>
</table>

**Example:** To divide `R[%eax]` by `R[%ecx]` do the following:

```
cld
idivl %ecx
movl %edx, rem
movl %eax, quot
```
Shift and Rotate Instructions

- **Right Shift**
  - Shift bits of operand one position to right
  - What happens to the most significant bit?
    - **Logical right shift** – msb filled with zero
      Ex: $10111011 \rightarrow 01011101$ ($187_{10} \rightarrow 93_{10}$)
        - If operand is treated as an **unsigned** integer; divides operand by 2, rounded down
    - **Arithmetic right shift** – msb retains previous value
      Ex: $10111011 \rightarrow 11011101$ ($-69_{10} \rightarrow -35_{10}$)
        - If operand is treated as a **signed** integer; divides operand by 2, rounded down
Shift and Rotate Instructions

**Left Shift**

- Shift bits of operand one position to left
- What happens to the least significant bit?
  - Least significant bit filled with zero
  - Ex: $01011011 \rightarrow 10110110$

- Some ISA's (including IA32) have both **logical** and **arithmetic left shift** instructions, but they do exactly same thing, as above.

- If operand is treated as an integer (signed or unsigned), left shift multiplies operand by 2 as long as there is no overflow.

  - Ex 1: $01011011 \rightarrow 10110110$ (as unsigned: $91_{10} \rightarrow 182_{10}$)
  - Ex 2: $01011011 \rightarrow 10110110$ (as signed: $+91_{10} \rightarrow -74_{10}$)

OVERFLOW! – sign reversal
Shift and Rotate Instructions

- **Right Rotate**
  - Shift bits of operand one position to right
  - Bit shifted out becomes the new msb
  - Ex: 10111010 → 01011101

- **Left Rotate**
  - Shift bits of operand one position to left
  - Bit shifted out becomes the new lsb
  - Ex: 10111010 → 01110101
Shift and Rotate Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shr &lt;items&gt;</td>
<td>Logical right shift &lt;items&gt; by &lt;items&gt; bits</td>
</tr>
<tr>
<td>sar &lt;items&gt;</td>
<td>Arithmetic right shift &lt;items&gt; by &lt;items&gt; bits</td>
</tr>
<tr>
<td>shl &lt;items&gt;</td>
<td>Logical left shift &lt;items&gt; by &lt;items&gt; bits</td>
</tr>
<tr>
<td>sal &lt;items&gt;</td>
<td>Arithmetic left shift &lt;items&gt; by &lt;items&gt; bits; same effect as shr &lt;items&gt;</td>
</tr>
<tr>
<td>ror &lt;items&gt;</td>
<td>Rotate right &lt;items&gt; by &lt;items&gt; bits</td>
</tr>
<tr>
<td>rol &lt;items&gt;</td>
<td>Rotate left &lt;items&gt; by &lt;items&gt; bits</td>
</tr>
</tbody>
</table>

- The source can either be an immediate constant with value between 0 and 31 inclusive, or the 1-byte register %cl (only the low-order 5 bits of %cl are used for the shift amount).

- Examples:

  ```
  movl $0xaabbccddd,%eax  # R[%eax] = 0xaabbccddd
  shll $4,%eax            # R[%eax] = 0xabbccddd0
  sarl $2,%eax            # R[%eax] = 0xeaef3374
  roll $4,%eax            # R[%eax] = 0xeaef3374e
  ```