# Machine-Level Programming: Basics

Arquitectura de Computadores
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

# Today: Machine Programming: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, moving
- Arithmetic & logical operations
- Intro to x86-64

# Definitions

- **Architecture: (also ISA: instruction set architecture) the parts of a processor design that one needs to understand to write assembly code**
  - The contract between the programmer and the hardware designer
  - Examples: instruction set specification, registers

- **Example ISAs:**
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: used in almost all mobile phones
  - RISC V: new open-source ISA

# Definitions

- **Microarchitecture: Implementation of the architecture**
  - Examples: pipelining, out-of-order execution, cache sizes, core frequency, …

- **Machine code: The byte-level programs that a processor executes**

- **Assembly code: A text representation of machine code**

# Intel x86 processors

- **Dominate laptop/desktop/server market**

- **Evolutionary design**
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on

- **Complex instruction set computer (CISC)**
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed, less so for low power

# Intel x86 evolution: Milestones

| *Name* | *Date* | *Transistors* | *MHz* |
|---|---|---|---|
| **8086** | **1978** | **29K** | **5-10** |

- First 16-bit Intel processor
- Basis for IBM PC & DOS, 1MB address space

| | | | |
|---|---|---|---|
| **386** | **1985** | **275K** | **16-33** |

- First 32-bit Intel processor, referred to as IA32
- Added "flat addressing", capable of running Unix

| | | | |
|---|---|---|---|
| **Pentium 4E** | **2004** | **125M** | **2800-3800** |

- First 64-bit Intel x86 processor, referred to as x86-64

| | | | |
|---|---|---|---|
| **Core 2** | **2006** | **291M** | **1060-3500** |

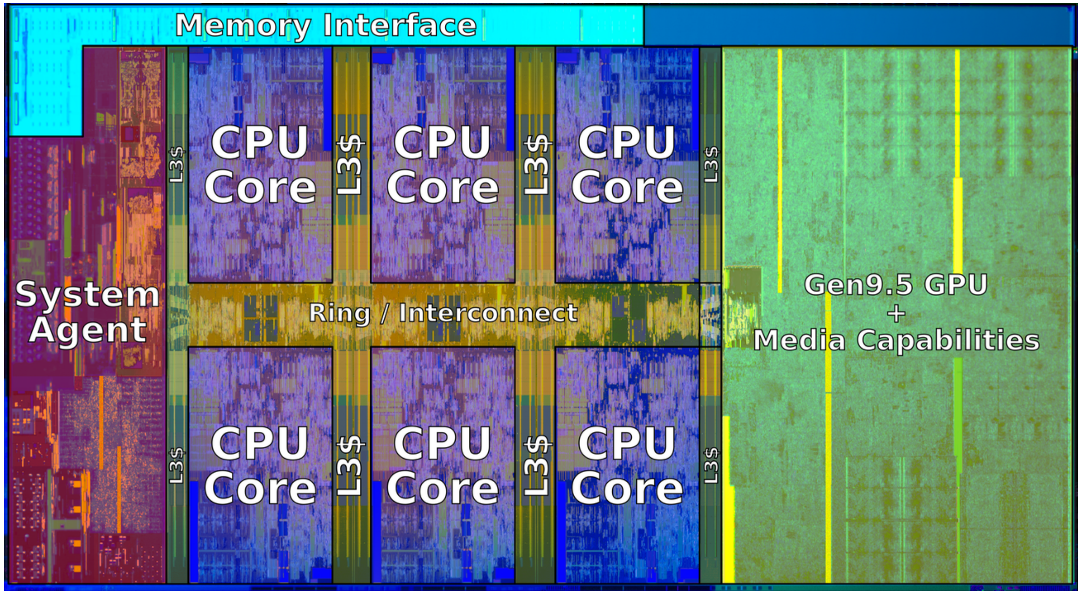
- First multi-core Intel processor

| | | | |
|---|---|---|---|
| **Core i7** | **2008** | **731M** | **1700-3900** |

- Four cores

# 2018 State of the Art: Coffee Lake

- **Mobile Model: Core i7**
  - 2.2-3.2 GHz
  - 45 W
- **Desktop Model: Core i7**
  - Integrated graphics
  - 2.4-4.0 GHz
  - 35-95 W
- **Server Model: Xeon E**
  - Integrated graphics
  - Multi-socket enabled
  - 3.3-3.8 GHz
  - 80-95 W



# Our coverage

- **Why only IA32?**
  - Most machines (even phones!) are 64-bit these days
  - x86-64 may be simpler than IA32 for user code

- **However...**
  - x86-64 is *not* simpler for kernel code
  - x86-64 is *not* simpler during debugging
    - More registers means more registers to have wrong values
  - x86-64 virtual memory is a bit of a drag
    - More steps than IA32, but not more intellectually stimulating
  - There are still a lot of 32-bit machines in the world
    - ...which can boot and run your personal OS

# Today: Machine Programming: Basics

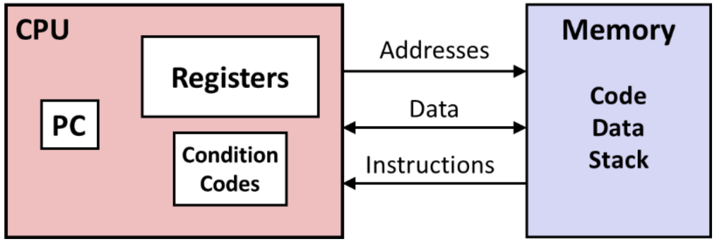
- History of Intel processors and architectures
- **C, assembly, machine code**
- Assembly Basics: Registers, operands, moving
- Arithmetic & logical operations
- Intro to x86-64

# Levels of abstraction

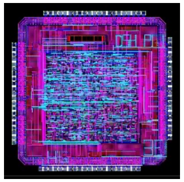
**C programmer**

```
int sum(int x, int y){
  int t = x+y;
  return t;
}
```

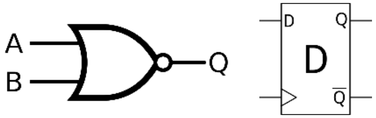
**Assembly programmer**




**Computer designer**

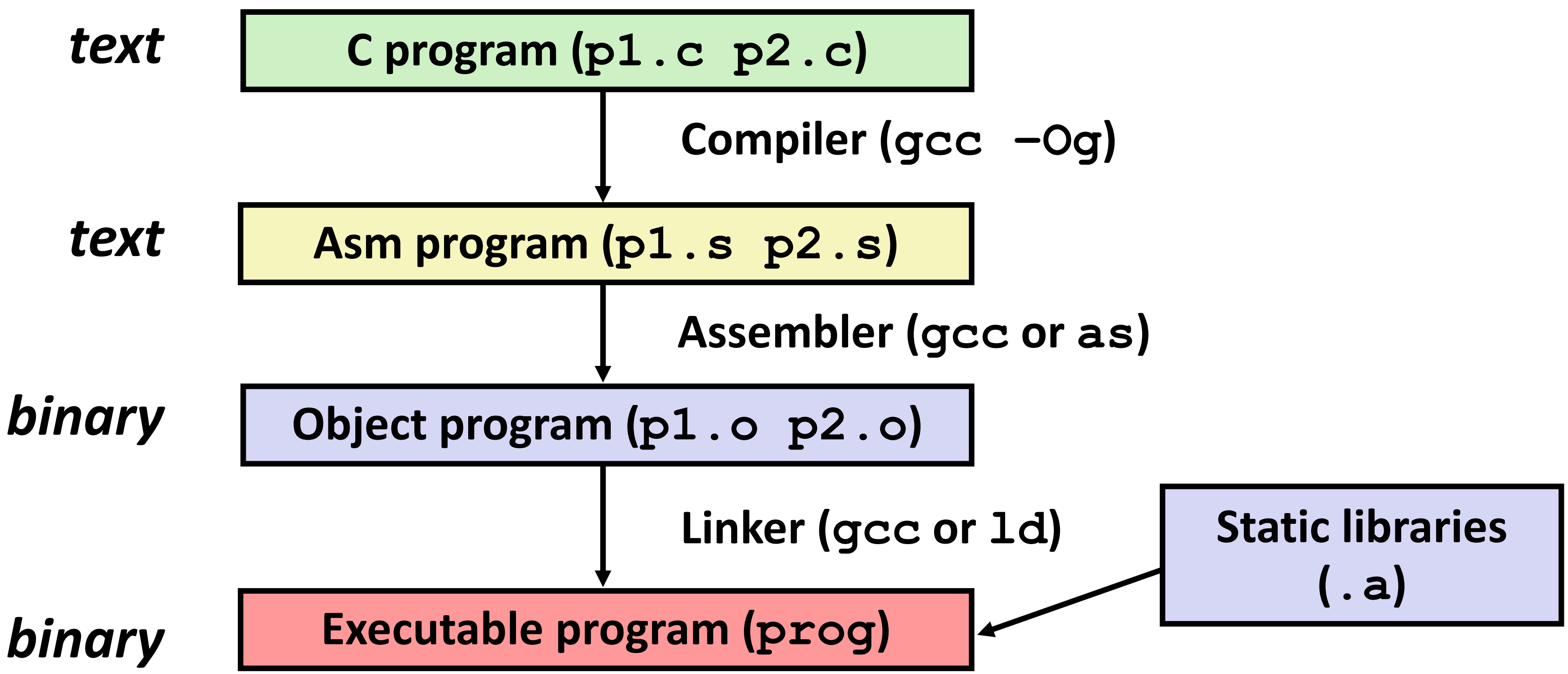


**Gates, clocks, circuit layout, …**



# Turning C into object code

- `gcc –Og p1.c p2.c -o prog`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]



# Compiling into Assembly

- `gcc –Og –S code.c`

## C code (*code.c*)

```
int sum(int x, int y){
  int t = x+y;
  return t;
}
```

## Generated IA32 Assembly

```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp, %esp
    popl %ebp
    ret
```

Set Up

Clean Up

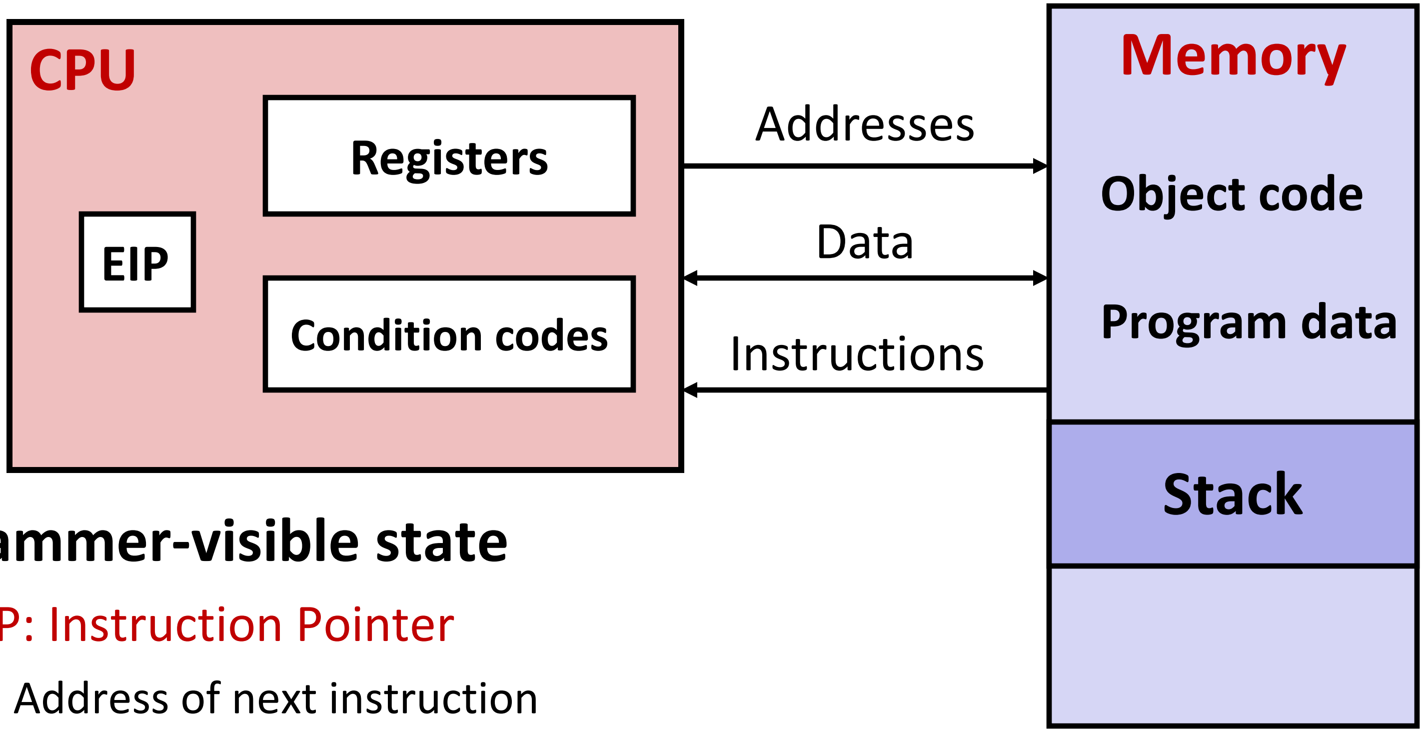
Some compilers use instruction "`leave`"

# Assembly functions

- **When a function executes, it needs to perform some setup/cleanup code**
  - That we call prologue (at the beginning) and epilogue (at the end)
  - Later, we will see why they are needed

- **While some functions *may* behave correctly with no epilogue and prologue, you should have them in all functions you write**

# Assembly programmer's view




## Programmer-visible state

- EIP: Instruction Pointer
  - Address of next instruction
- Registers
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

## Memory

- Byte addressable array
- Code and user data
- Stack to support procedures

# Assembly characteristics: Data types

- **Integer data of 1, 2, 4, 8 or 16 bytes**
  - Data values
  - Addresses (untyped pointers)

- **Floating point data of 4, 8, or 10 bytes**

- **Code**
  - Byte sequences encoding series of instructions
  - IA32 instructions can range in length from 1 to 15 bytes

- **No aggregate types such as arrays or structures**
  - Just contiguously allocated bytes in memory

# Assembly characteristics: Operations

- **Perform arithmetic function on register or memory data**

- **Transfer data between memory and register**
  - Load data from memory into register
  - Store register data into memory

- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches

# Machine code

- **Assembler**
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files

- **Linker**
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for **`malloc`**, **`printf`**
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

# Machine code

## C code (*code.c*)

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

## Generated IA32 Assembly

```
sum:
   pushl %ebp
   movl %esp,%ebp
   movl 12(%ebp),%eax
   addl 8(%ebp),%eax
   movl %ebp, %esp
   popl %ebp
   ret
```

## Machine code for `sum`

```
0x401040 <sum>:
   0x55
   0x89
   0xe5
   0x8b
   0x45
   0x0c
   0x03
   0x45
   0x08
   0x5d
   0xc3
```

- Total of 11 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address **`0x401040`**

# Machine instruction example

```
int t = x+y;
```

- **C code**
  - Add two signed integers

```
addl 8(%ebp),%eax
```

**Similar to expression:**

```
x += y
```

**More precisely:**

```
int eax;
int *ebp;
eax += ebp[2]
```

- **Assembly**
  - Add two 4-byte integers
  - Operands:

| | | |
|---|---|---|
| **x:** | Register | **%eax** |
| **y:** | Memory | **M[%ebp+8]** |
| **t:** | Register | **%eax** |

– Return function value in **%eax**

```
0x80483ca:   03 45 08
```

- **Object code**
  - 3-byte instruction
  - Stored at address **0x80483ca**

# Disassembling machine code

## Disassembled

```
080483c4 <sum>:
 80483c4:  55          pushl   %ebp
 80483c5:  89 e5       movl    %esp,%ebp
 80483c7:  8b 45 0c    movl    0xc(%ebp),%eax
 80483ca:  03 45 08    addl    0x8(%ebp),%eax
 80483cd:  5d          popl    %ebp
 80483ce:  c3          ret
```

- **Disassembler: `objdump -d prog`**
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

# Alternate disassembly

**Object**

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x5d
    0xc3
```

**Disassembled**

```
Dump of assembler code for function sum:
0x080483c4 <sum+0>:     pushl   %ebp
0x080483c5 <sum+1>:     movl    %esp,%ebp
0x080483c7 <sum+3>:     movl    0xc(%ebp),%eax
0x080483ca <sum+6>:     addl    0x8(%ebp),%eax
0x080483cd <sum+9>:     popl    %ebp
0x080483ce <sum+10>:    ret
```

- **Within gdb debugger**

  **`gdb prog`**

  **`disassemble sum`**
  - Disassemble procedure

  **`x/11xb sum`**
  - Examine the 11 bytes starting at `sum`

# What can be disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:    file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55             push   %ebp
30001001:  8b ec          mov    %esp,%ebp
30001003:  6a ff          push   $0xffffffff
30001005:  68 90 10 00 30 push   $0x30001090
3000100a:  68 91 dc 4c 30 push   $0x304cdc91
```

- **Anything that can be interpreted as executable code**
- **Disassembler examines bytes and reconstructs assembly source**

# Today: Machine Programming: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- **Assembly basics: Registers, operands, moving**
- Arithmetic & logical operations
- Intro to x86-64

# Integer registers (IA32)

general purpose

| 32-bit | 16-bit | High byte | Low byte | Origin (mostly obsolete) |
|---|---|---|---|---|
| %eax | %ax | %ah | %al | *accumulate* |
| %ecx | %cx | %ch | %cl | *counter* |
| %edx | %dx | %dh | %dl | *data* |
| %ebx | %bx | %bh | %bl | *base* |
| %esi | %si | | | *source index* |
| %edi | %di | | | *destination index* |
| %esp | %sp | | | ***stack pointer*** |
| %ebp | %bp | | | ***base pointer*** |

16-bit virtual registers
(backwards compatibility)

# Moving data

- `mov` ***source, destination***

- **Operand types**
  - ***Immediate:*** Constant integer data
    - Examples: **`$0x400, $-533`**
    - Encoded with 1, 2, or 4 bytes (**`b,w,l`**)

  - ***Register:*** One of 8 integer registers
    - **`%esp`** and **`%ebp`** reserved for special use
    - Others have special uses for particular instructions

  - ***Memory:*** 1, 2 or 4 consecutive bytes of at address given by register
    - Simplest example: **`(%eax)`**
    - Various other address modes

# mov operand combinations

| | Source | Dest | Example | C Analogy |
|---|---|---|---|---|
| mov | *Imm* | *Reg* | `movl $0x4,%eax` | `temp = 0x4;` |
| | | *Mem* | `movl $-147,(%eax)` | `*p = -147;` |
| | *Reg* | *Reg* | `movl %eax,%edx` | `temp2 = temp1;` |
| | | *Mem* | `movl %eax,(%edx)` | `*p = temp;` |
| | *Mem* | *Reg* | `movl (%eax),%edx` | `temp = *p;` |

***Cannot do memory-memory transfer with a single instruction***

# Simple memory addressing modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address

```
movl (%ecx),%eax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movl 8(%ebp),%edx
```

# Today: Machine Programming: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly basics: Registers, operands, moving
- **Arithmetic & logical operations**
- Intro to x86-64

# Some arithmetic operations

- **Two operand instructions**

| Format | | Computation | |
|---|---|---|---|
| `add` | Src,Dest | Dest = Dest + Src | |
| `sub` | Src,Dest | Dest = Dest – Src | |
| `imul` | Src,Dest | Dest = Dest * Src | (signed mul) |
| `sal` | Src,Dest | Dest = Dest << Src | (same as *shll*) |
| `sar` | Src,Dest | Dest = Dest >> Src | (arithmetic shift) |
| `shr` | Src,Dest | Dest = Dest >> Src | (logical shift) |
| `xor` | Src,Dest | Dest = Dest ^ Src | |
| `and` | Src,Dest | Dest = Dest & Src | |
| `or` | Src,Dest | Dest = Dest \| Src | |

# Some arithmetic operations

- **One operand instructions**

| Format | | Computation | |
|---|---|---|---|
| `inc` | Dest | Dest = Dest + 1 | |
| `dec` | Dest | Dest = Dest – 1 | |
| `neg` | Dest | Dest = – Dest | |
| `not` | Dest | Dest = ~Dest | |
| `mul` | Src | `%edx:%eax` = Src * `%eax` | (unsigned) |
| `div` | Src | `%eax` = `%edx:%eax` / Src | (unsigned) |
| `idiv` | Src | `%eax` = `%edx:%eax` / Src | (signed) |

# Some arithmetic operations

- **Watch out for argument order**

- **Most instructions do not make any distinction between *signed* and *unsigned***
  - Exceptions are multiplication and division (why?)

- **See reference manual for more instructions**
  - http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
  - Note: it uses the Intel syntax rather than the AT&T syntax, typically used in Unix

# Arithmetic expression example

**C code**

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

**Assembly code**

```
logical:
    pushl %ebp              } Set Up
    movl %esp,%ebp

    movl 12(%ebp),%eax      } Body
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp, %esp         } Finish
    popl %ebp
    ret
```

```
movl 12(%ebp),%eax     # eax = y
xorl 8(%ebp),%eax      # eax = x^y          (t1)
sarl $17,%eax          # eax = t1>>17       (t2)
andl $8185,%eax        # eax = t2 & mask (rval)
```

# Understanding *logical()*

## C code

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

## Assembly code

```
logical:
    pushl %ebp            } Set
    movl %esp,%ebp        } Up

    movl 12(%ebp),%eax    }
    xorl 8(%ebp),%eax     } Body
    sarl $17,%eax         }
    andl $8185,%eax       }

    movl %ebp, %esp       }
    popl %ebp             } Finish
    ret                   }
```

```
movl 12(%ebp),%eax    # eax = y
xorl 8(%ebp),%eax     # eax = x^y          (t1)
sarl $17,%eax         # eax = t1>>17       (t2)
andl $8185,%eax       # eax = t2 & mask    (rval)
```

# Understanding *logical()*

## C code

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

## Assembly code

```
logical:
    pushl %ebp                    Set
    movl %esp,%ebp                Up

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax             Body
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp, %esp
    popl %ebp                     Finish
    ret
```

```
movl 12(%ebp),%eax     # eax = y
xorl 8(%ebp),%eax      # eax = x^y          (t1)
sarl $17,%eax          # eax = t1>>17       (t2)
andl $8185,%eax        # eax = t2 & mask (rval)
```

# Understanding *logical()*

## C code

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192$

$2^{13} - 7 = 8185$

## Assembly code

```
logical:
    pushl %ebp            } Set Up
    movl %esp,%ebp

    movl 12(%ebp),%eax    
    xorl 8(%ebp),%eax     } Body
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp, %esp       
    popl %ebp             } Finish
    ret
```

```
movl 12(%ebp),%eax    # eax = y
xorl 8(%ebp),%eax     # eax = x^y         (t1)
sarl $17,%eax         # eax = t1>>17      (t2)
andl $8185,%eax       # eax = t2 & mask   (rval)
```

# Understanding *logical()*

- **Some functions return a value, and that value must be received reliably by the function's caller**

- **Integral up to 32-bits (`char`, `short`, `int`, `long`, pointer)**
  - Store return value in `%eax`

- **Integral of 64-bits (`long long`)**
  - Store return value in `%edx:%eax`

- **Floating-point type**
  - Store return value in floating-point register `st(0)` (beyond scope of course)

# Today: Machine Programming: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, moving
- Arithmetic & logical operations
- Intro to x86-64

# Data representations: IA32 + x86-64

| C Data Type | Generic 32-bit | Intel IA32 | x86-64 |
|---|---|---|---|
| char | 1 | 1 | 1 |
| short | 2 | 2 | 2 |
| int | 4 | 4 | 4 |
| long | 4 | 4 | 8 |
| float | 4 | 4 | 4 |
| double | 8 | 8 | 8 |
| long double | 8 | 10/12 | 16 |
| char * | 4 | 4 | 8 |

– Or any other pointer

# x86-64 Integer registers

- Extends existing registers and adds 8 new ones

| | | | |
|---|---|---|---|
| %rax | %eax | %r8 | %r8d |
| %rbx | %ebx | %r9 | %r9d |
| %rcx | %ecx | %r10 | %r10d |
| %rdx | %edx | %r11 | %r11d |
| %rsi | %esi | %r12 | %r12d |
| %rdi | %edi | %r13 | %r13d |
| %rsp | %esp | %r14 | %r14d |
| %rbp | %ebp | %r15 | %r15d |

# Instructions

- **Long word `l` (4 Bytes) ↔ Quad word `q` (8 Bytes)**

- **New instructions:**
  - **`movl` → `movq`**
  - **`addl` → `addq`**
  - **`sall` → `salq`**
  - etc.

- **32-bit instructions that generate 32-bit results**
  - Set higher order bits of destination register to `0`
  - Example: **`addl`**

# 32-bit code for *swap()*

**C code**

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

**Assembly code**

```
swap:
   pushl %ebp
   movl  %esp,%ebp
   pushl %ebx

   movl   8(%ebp),%edx
   movl   12(%ebp),%ecx
   movl   (%edx),%ebx
   movl   (%ecx),%eax
   movl   %eax,(%edx)
   movl   %ebx,(%ecx)

   popl   %ebx
   popl   %ebp
   ret
```

Set Up

Body

Finish

# 64-bit code for *swap()*

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:                              Set Up

   movl    (%rdi),%edx
   movl    (%rsi),%eax             Body
   movl    %eax,(%rdi)
   movl    %edx,(%rsi)

   ret                             Finish
```

- **Operands passed in registers**
  - First (**`xp`**) in **`%rdi`**, second (**`yp`**) in **`%rsi`**
  - 64-bit pointers
  - No stack operations required
- **32-bit data**
  - Data held in registers **`%eax`** and **`%edx`**
  - **`movl`** operation

# 64-bit code for *long int swap()*

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap_l:
                                  Set Up

   movq   (%rdi),%rdx
   movq   (%rsi),%rax             Body
   movq   %rax,(%rdi)
   movq   %rdx,(%rsi)

   ret                            Finish
```

- **64-bit data**
  - Data held in registers **`%rax`** and **`%rdx`**
  - **`movq`** operation
    - "q" stands for quad-word

# Machine Programming: Basics: Summary

- **History of Intel processors and architectures**
  - Evolutionary design leads to many quirks and artifacts

- **C, assembly, machine code**
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences

- **Assembly Basics: registers, operands, *mov***
  - The x86 *mov* instruction covers a wide range of data movement forms

# Machine Programming: Basics: Summary

- **Arithmetic**
  - C compiler will figure out different instruction combinations to carry out computation

- **Intro to x86-64**
  - A major departure from the style of code seen in IA32