

# CSC2/452 Computer Organization

## Addresses and Instructions

Sreepathi Pai

URCS

September 21, 2022

# Outline

Administrivia

Bit-Level Universe: A Recap

Addresses

Instruction Set Architecture

# Outline

Administrivia

Bit-Level Universe: A Recap

Addresses

Instruction Set Architecture

# Announcements

- ▶ Homework #2 due today IN CLASS.
- ▶ Assignment #1 out
  - ▶ Due next week, Monday Sep 26.
- ▶ Homework #3 is out today
  - ▶ Due next Wed as usual

# Outline

Administrivia

Bit-Level Universe: A Recap

Addresses

Instruction Set Architecture

# Bits

- ▶ Bits: 0 and 1
  - ▶ Bit operations: AND, OR, NOT, XOR, etc.
- ▶ Data structures made of bits
  - ▶ Bitsets (also called Bitmaps, Bitstrings, Bitvectors, etc.)
  - ▶ Bitfields
  - ▶ Bitwise operations: AND, OR, NOT, XOR, ...
  - ▶ Primary operations: setting bits, extracting bits, shifting bits
- ▶ Applications
  - ▶ Representing integers (bitfield of sign + value)
  - ▶ Representing floats: single-precision, etc.

## What is this bitstring?

1100 0000 1000 1001 0000 1111 1101 1100

- ▶ Hexadecimal value: `0xc0890fdb`
- ▶ A: The 32-bit unsigned integer: 3,226,013,659
- ▶ B: The 32-bit signed two's complement integer: -1,068,953,637
- ▶ C: The 32-bit IEEE754 single-precision float: -3.141593
- ▶ D: All of the above
- ▶ E: None of the above

# Bits are in the eye of the beholder

- ▶ If the ALU consumes the bits, it is interpreted as an integer
  - ▶ either signed or unsigned depending on the context (more later)
- ▶ If the FPU consumes the bits, it is interpreted as a float
- ▶ Bits themselves do not encode what they mean
  - ▶ Powerful idea
  - ▶ Good sometimes, mostly bad?
- ▶ High-level languages introduce the notion of *type safety* to maintain logical sanity
  - ▶ Roughly, track what the bits mean and prevent invalid operations



# Basic Machine Level Data “Types”

- ▶ Integers
- ▶ Floats
- ▶ Addresses
- ▶ Instructions

# Outline

Administrivia

Bit-Level Universe: A Recap

Addresses

Instruction Set Architecture

# Addresses

- ▶ An address is ultimately an unsigned integer
- ▶ It represents a location in memory (i.e. RAM)
- ▶ Addresses on a machine are usually of a fixed size
  - ▶ 16-bit (a long time ago, and in some very small machines these days)
  - ▶ 32-bit (remarkably common until about a decade ago)
  - ▶ 64-bit (most common today)

# The Size of an Address

- ▶ Addresses run from 0 to  $2^n - 1$  where  $n$  is 16, 32, or 64-bit
  - ▶ 16: 65,536
  - ▶ 32: 4,294,967,296
  - ▶ 64: 18,446,744,073,709,551,616
- ▶ If machines are byte addressable (as most are), this means the *maximum* size of addressable memory is:
  - ▶ 16: approx. 64 Kilobytes
  - ▶ 32: approx. 4 Gigabytes
  - ▶ 64: approx. 18 Exabytes
- ▶ Most *single* computers these days can accommodate upto a few terabytes
  - ▶ Intel and AMD 64-bit hardware only supports 48 bits (about 256 Terabytes)

# Creating Addresses

- ▶ Addresses are just integers *at the machine level*
  - ▶ High-level languages try very hard to prevent integers and addresses from mixing
- ▶ Two primary uses of addresses
  - ▶ Address of code
  - ▶ Address of data
  - ▶ No way to distinguish between these uses by looking at only the address without additional info
- ▶ Three primary address consuming units inside a CPU
  - ▶ The *instruction fetch* unit treats address as address of code
  - ▶ The *load/store* unit treats address as address of data
  - ▶ The *address generation* unit performs simple operations on addresses

## Addresses in Assembly: Absolute Addresses

```
callABS    printf    /* printf is an absolute address  
                    of the printf code */
```

- ▶ (Recall that labels are just human-readable addresses)
- ▶ Here the address of `printf` would be filled in by the linker
- ▶ It is a full 64-bit address
- ▶ This is calling the `printf` function by providing its address
- ▶ In machine code, this could be represented `XX 00 00 00 00 00 00 00`, where the zeroes are filled in with the 64-bit address and `XX` represents the opcode.
- ▶ Post lecture update: x86 does not permit absolute addresses as constants for its `call` instruction, so the `callABS` instruction is made up.

## Addresses in Assembly: Indirect Addresses

```
call    *%rdx
```

- ▶ Here, `%rdx` is a 64-bit register containing an address
  - ▶ We'll talk about registers later, but assume it is a special, named memory location inside the CPU
  - ▶ Or, in high-level language terms, it is a variable
- ▶ This is calling *whatever* is at the address contained in `%rdx`
- ▶ In machine code, this is represented as `ff d2`, note no address
- ▶ The `*%rdx` syntax is AT&T, you may also see `call [rdx]` in Intel syntax

## Operations on Indirect Address Operands

- ▶ An *effective address* (EA) is the final address formed from an indirect address and (optional) various indexing and scaling operations.

Expression	Example	Meaning
<code>%reg</code>	<code>%rdx</code>	EA is in <code>%reg</code>
<code>disp(%reg)</code> displacement is a 8/16/32 bit signed integer	<code>-4(%rbp)</code>	EA is <code>%reg + disp</code>
<code>(%basereg, %indexreg, scale)</code> scale can only be 1, 2, 4, or 8	<code>(%rbx,%rsi,4)</code>	EA = <code>%basereg + %indexreg * scale</code>
<code>disp(%basereg, %indexreg, scale)</code>	<code>-8(%rbx,%rsi,4)</code>	EA = <code>%basereg + %indexreg * scale + disp</code>

- ▶ These are Intel 64 assembly language *addressing modes*, represented in AT&T syntax



# Computing on Addresses

```
.LC1:
    .string    "Hello, the value of pi is %f\n"
    ...
    leaq      .LC1(%rip), %rdi
    ...
```

- ▶ `.LC1` is a label (i.e., an address)
- ▶ `leaq` is *Load Effective Address Quadword*
  - ▶ Quadword is 64-bit
- ▶ Compute `%rip + .LC1` and store it in `%rdi`
  - ▶ After assembly, `.LC1` is `0x99`
  - ▶ So the machine code looks like `leaq 0x99(%rip), %rdi`
  - ▶ During execution, `%rdi = %rip + 0x99`
  - ▶ This calculation performed by the address generation unit (AGU) [not the ALU]
  - ▶ `%rip` is the register that holds the address of the next instruction to be executed

## Base, Index, Scale and Displacement

- ▶ The `disp(%basereg, %indexreg, scale)` addressing mode is most helpful for calculating addresses of array elements

```
/* pseudocode to find address of array element 5 */  
/* array Arr contains doubles, recall doubles are 64-bit*/  
array_base = Arr  
index = 5  
address_of_fifth = array_base + index * 8
```

- ▶ In pseudo-assembly:

```
/* moving absolute address into %rbx */  
leaq Arr, %rbx  
  
movq $5, %rsi  
  
/* compute address of fifth element */  
leaq (%rbx, %rsi, 8), %rdx  
  
/* (contd. on next slide */
```

## Loading data using an effective address

```
/* %rdx has address of fifth element, load the element into %rax */  
movq (%rdx), %rax
```

- ▶ The `mov` instruction is the primary way to load or store data on Intel 64
  - ▶ Note: It copies data, does not move it
- ▶ `mov` as several forms
  - ▶ `mov %REG1, %REG2`, copies data in `%REG1` to `%REG2`
  - ▶ `mov IMM, %REG`, copies constant IMMEDIATE to `%REG`
  - ▶ `mov EA_EXPR, %REG`, load data from effective address `EA_EXPR` into `%REG`
  - ▶ `mov %REG, EA_EXPR`, store data from `%REG` to effective address computed by `EA_EXPR`
- ▶ `EA_EXPR` are all the forms we saw previously: e.g. `0x99(%rip)`, etc.
- ▶ Accesses to memory are performed by the load/store unit

## Addresses for data: Data size

- ▶ Addresses do not contain information about the *size* of data they address
- ▶ It is possible to use the same address  $x$  to read:
  - ▶ 1 byte stored at  $x$
  - ▶ 2 bytes at  $x$  and  $x + 1$
  - ▶ 4 bytes at  $x, x + 1, x + 2, x + 3$
  - ▶ 8 bytes at  $x, x + 1, x + 2, x + 3, \dots, x + 7$
- ▶ In AT&T syntax, the size of data is indicated by a suffix on the instruction:
  - ▶ `movq` (quad, 64-bit integer)
  - ▶ `movl` (long, 32-bit integer, 64-bit floating point)
  - ▶ `movw` (word, 16-bit integer)
  - ▶ `movb` (byte, 8-bit byte)
  - ▶ `movs` (single, 32-bit float)

## Addresses for data: Alignment

- ▶ An address is set to be aligned to be  $x$  if it is divisible by  $x$
- ▶ For example,
  - ▶ All even addresses are aligned to 2
  - ▶ Address  $0x44$  is aligned to 4 (lowest two bits are zero)
- ▶ On many machines, data of size  $n$  can only be read if it stored at address  $x$  aligned to  $n$ 
  - ▶ No such general alignment requirement on Intel/AMD machines
  - ▶ Some instructions may have these requirements on Intel/AMD
- ▶ Attempting to read misaligned data is
  - ▶ slower on Intel/AMD
  - ▶ can cause errors on other processors

# Relative Addresses

```
610:    eb 06                jmp     618 <main+0x1e>
```

- ▶ Recall output of `objdump -S`
  - ▶ Columns contain: address, machine code, disassembly
  - ▶ `jmp` instruction at address `0x610`
  - ▶ encoded as two bytes `eb 06`
  - ▶ `jmp` to address `0x618`
- ▶ Is address `0x618` stored in the machine code?

## Relative Addresses (contd.)

- ▶ Some addresses can be expressed as relative to an *implicit* register
  - ▶ usually `%rip`
- ▶ In the previous slide, *after* reading the `jmp` instruction
  - ▶ `%rip` is `0x612`
- ▶ The instruction contains an offset (`0x6`) that is added to `%rip`
  - ▶ i.e., it behaves like `jmp 0x6(%rip)`
- ▶ Sometimes called a short jump or a near jump

# Implicit Addressing

```
pushq %rbp
```

- ▶ Some instructions use *implicit* indirect addresses
- ▶ On x86, notably `pushq`, `popq`, `leave`
  - ▶ These store values on the function stack in memory
- ▶ The address of the function stack is given by register `%rsp`
- ▶ Executing `pushq %rbp` is equivalent to:

```
%rsp = %rsp - 8    /* note stack grows towards lower address */  
(%rsp) = %rbp     /* note (%rsp) is indirect memory address */
```



# Hello Pi, again

```
.file "hellopi.c"
.text
.section .rodata
.LC1:
.string "Hello, the value of pi is %f\n"
.text
.globl main
.type main, @function
main:
.LFB0:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movss .LC0(%rip), %xmm0
movss %xmm0, -4(%rbp)
cvtss2sd -4(%rbp), %xmm0
leaq .LC1(%rip), %rdi
movl $1, %eax
call printf@PLT
movl $0, %eax
leave
ret
.section .rodata
.align 4
.LC0:
.long 1078530011
```

LC1 is address of this string

main is address of the next instruction  
so is .LFB0  
store the value of %rbp at (%rsp)  
copy (NOT move) the value of %rsp into %rbp  
subtract 16 from %rsp  
load single at .LC0(%rip) into %xmm0  
store single in %xmm0 into -4(%rbp)  
convert single at -4(%rbp) into double in %xmm0  
load address of string into %rdi  
set %eax to 1  
call printf  
set %eax to 0  
set %rsp to %rbp, then popq %rbp  
exit the function

make sure .LC0 is aligned to 4

this is 0x40490fdb, i.e. 3.141593

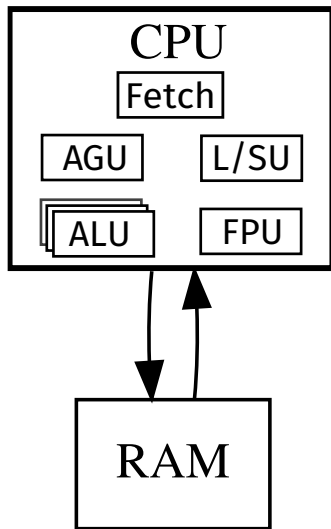
## Unexplained stuff (for now)

- ▶ Standard x86 function prologue and epilogue
  - ▶ `push %rbp, %rsp` (we know what it does, but why?)
  - ▶ `subq $16, %rsp`
  - ▶ `leave`
- ▶ C translation details
  - ▶ `cvtss2sd -4(%rbp), %xmm0` (why convert to double?)
  - ▶ `movl $1, %eax`
- ▶ Details on these as we study translating C to assembly in later classes

# Food for thought

Why do we need so many ways to specify an address?

# The Computer (Updated)



# Outline

Administrivia

Bit-Level Universe: A Recap

Addresses

Instruction Set Architecture

# Instructions

- ▶ “Commands” to the processor (e.g. CPU)
- ▶ Part of Instruction Set Architecture
  - ▶ *Programmer’s interface to a processor*
  - ▶ Instructions processor understands
  - ▶ Data types processor understands
  - ▶ Other processor services and functionality [later in the course]

# Design Questions

- ▶ Which instructions should a processor support?
- ▶ Where are inputs and outputs for instructions stored?

# Functionality

- ▶ Bitwise manipulation instructions
- ▶ Integer arithmetic
- ▶ Floating point arithmetic
- ▶ Address manipulation instructions
- ▶ Memory load/store instructions
- ▶ Function call instructions
- ▶ Stack manipulation instructions
- ▶ Cryptography instructions?
- ▶ Video encoding/decoding instructions?
- ▶ ...



# CISC vs RISC

- ▶ Complex Instruction Set Computer (CISC)
  - ▶ Supports many instructions
- ▶ Reduced Instruction Set Computer (RISC)
  - ▶ Supports a limited number of instructions

## CISC vs RISC (contd.)

CISC	RISC
Easier to program by hand Complex circuit implementation Can be made fast VAX, Intel x86	Easier for compilers to program Simpler circuit implementation Can be made fast (usually easier) MIPS (most famous), ARM, RISC-V, ...

- ▶ Distinction is somewhat meaningless now
  - ▶ Unlikely it ever was meaningful
- ▶ Intel x86 is internally RISC
  - ▶ Instructions you use are translated into RISC-like micro-instructions
- ▶ ARM (Acorn RISC Machines) is hardly RISC anymore
  - ▶ Lots of instructions supported

# Operands

- ▶ Instructions operate on *operands*
- ▶ Operands can be stored in:
  - ▶ Registers
  - ▶ Memory
- ▶ Should all instructions be allowed to access both kinds of operands?

# Load/Store architectures

- ▶ Should all instructions be allowed to access both kinds of operands?
- ▶ Yes
  - ▶ All instructions can access both memory and register operands
  - ▶ Mostly CISC
- ▶ No
  - ▶ Only load/store instructions can access memory and registers
  - ▶ All other instructions can access only registers
  - ▶ Mostly RISC
- ▶ VAX
  - ▶ Fully orthogonal architecture
  - ▶ Instructions and operand accesses are independent
  - ▶ Makes it much easier to program
- ▶ x86
  - ▶ Most instructions can access both memory and registers
  - ▶ But not orthogonal

# Designing a Instruction Encoding

- ▶ Consider a load/store architecture processor
- ▶ 128 instructions
- ▶ Supports 64-bit addresses
- ▶ Has 32 registers

# Encoding

OP RS1, RS2, RD

- ▶ 128 different instructions (i.e. OP can be 0 to 127)
  - ▶ How many bits?
- ▶ Has 32 registers
  - ▶ RS1, RS2 and RD are all registers
  - ▶ How many bits?

# Encoding (Solutions)

OP RS1, RS2, RD

- ▶ 128 different instructions (i.e. OP can be 0 to 127)
  - ▶ 7 bits
- ▶ Has 32 registers
  - ▶ RS1, RS2 and RD are all registers
  - ▶ 5 bits + 5 bits + 5 bits = 15 bits
- ▶ Can store all the information we need in 22 bits
  - ▶ Maybe use a 32-bit bitfield?

# Instructions as a bitfield

```
3322222222 22111 11111 11
1098765432|10987|65432|10987|6543210
-----
unused   |  RD  | RS2  | RS1  | opcode
-----
```

- ▶ Top two lines indicate bit positions
- ▶ Here is what ADD R0, R1, R2 looks like
  - ▶ assume ADD is given opcode  $0010001_2$
  - ▶ assume Rx is indicated by x in binary
  - ▶ i.e. R0 is  $00000_2$ , R20 is  $10100_2$
  - ▶ set unused bits to 0

```
3322222222 22111 11111 11
1098765432|10987|65432|10987|6543210
-----
0000000000|00010|00001|00000|0010001
-----
```



# Our Instruction Encoding

- ▶ Fixed-width instructions
  - ▶ All instructions 32-bit
- ▶ Not very flexible
  - ▶ Only register operands
  - ▶ Does not support advanced addressing modes
  - ▶ Can't supply constants (or *immediate* operands)

# Open Design Questions

- ▶ How do we encode instructions that take memory addresses?
  - ▶ Won't fit in 32 bits
- ▶ If an operation takes a memory address, what do we do with RS1, RS2 and RD fields?
- ▶ How do we specify data sizes?

# Intel x86 instruction coding

```

7          .globl  main
9          main:
10         .LFB0:
11 0000 55          pushq   %rbp
12 0001 4889E5      movq    %rsp, %rbp
13 0004 4883EC10    subq   $16, %rsp
14 0008 F30F1005    movss  .LC0(%rip), %xmm0
14         00000000
15 0010 F30F1145    movss  %xmm0, -4(%rbp)
15         FC
16 0015 F30F5A45    cvtss2sd -4(%rbp), %xmm0
16         FC
17 001a 488D3D00    leaq   .LC1(%rip), %rdi
17         000000
```

- ▶ Variable-sized
- ▶ Registers, Memory, Constants as operands
- ▶ Advanced addressing modes
- ▶ But also implemented as a bitfield!

# Intel Instruction Format

## B.1 MACHINE INSTRUCTION FORMAT

All Intel Architecture instructions are encoded using subsets of the general machine instruction format shown in Figure B-1. Each instruction consists of of:

- an opcode
- a register and/or address mode specifier consisting of the ModR/M byte and sometimes the scale-index-base (SIB) byte (if required)
- a displacement and an immediate data field (if required)

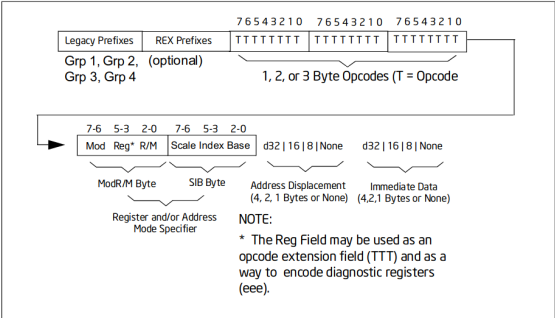


Figure B-1. General Machine Instruction Format

Source: Intel 64 and IA-32 Architectures: Software Developers Manual, Volume 2, Instruction Set Reference (A-Z), pg. 2095

# x86 Opcode distribution



Fraunhofer

FKIE

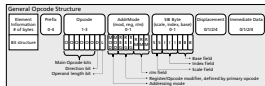
FRAUNHOFER-INSTITUT FÜR KOMMUNIKATION, INFORMATIONSVERARBEITUNG UND ERGONOMIE FKIE

## x86 Opcode Structure and Instruction Overview

Prefix	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		
	0	ADD				ES	ES					OR		CS			TWO BYTE	
	1	ADC				SS	SS					SBB		DS				
	2	AND				ES	DAA					SUB		CS		DAS		
	3	XOR				SS	AAA					CMP		DS		AAS		
	4	INC										DEC						
	5	PUSH										POP						
	6	BOUND	BOUND	BOUND	ARPL	FS	GS	OP_SIZE	OP_SIZE	PUSH	IMUL	PUSH	IMUL	INS		OUTS		
	7	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JJ	JJS	JPE	JPO	JL	JGE	JLE	JG	
	8	ADD/ADC/AND/XOR	OR/SBB/SUB/CMP		TEST	XCHG					MOV REG	MOV SREG	LEA	MOV SREG	POP			
	9	NOP	XCHG EAX				CWD	CDQ	CALL	WAIT	SCALD	SCALD	SAHF	LAHF				
	A	MOV EAX				MOV					CMP	TEST	STOS	LODS	SCAS			
	B	MOV																
	C	SHIFT IMM	RETN	LES	LDS	MOV IMM	ENTER	LEAVE	RETF	INT3	INT IMM	INTO	INTO	RETD				
	D	SHIFT 1	SHIFT CL	AAM	AAD	SALC	XLAT				FPU							
	E	LOCK	LOCK	LOCK	LOCK	IN IMM	OUT IMM				IN IMM	JMP	JMPF	JMP	IN DX	OUT DX		
	F	LOCK	ICE	BP	ADP	ADP	HLT	CMC	TEST/INSTR	EMUL/CDQ	CLC	STC	CLI	STI	CLD	STD	INC	DEC

Prefix	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
	0	LOCK	LOCK	LOCK	LOCK	LOCK	LOCK	LOCK	LOCK	LOCK	LOCK	LOCK	LOCK	LOCK	LOCK	LOCK	
	1	SSE(1,2,3)										HINT_NOP					
	2	MOV CR/DR										SSE(1,2)					
	3	WRMSR	RDTSC	RDMSR	WRMSR	WRMSR	WRMSR	WRMSR	WRMSR	WRMSR	WRMSR	WRMSR	WRMSR	WRMSR	WRMSR	WRMSR	
	4	CMOV															
	5	SSE(1,2)															
	6	MMX, SSE2															
	7	MMX, SSE(1,2,3), VMX															
	8	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JJ	JJS	JPE	JPO	JL	JGE	JLE	JG
	9	SETO	SETNO	SETB	SETNB	SETL	SETNL	SETLE	SETNLE	SETG	SETNG	SETGE	SETNGE	SETL	SETNL	SETLE	SETNLE
	A	PUSH	POP	CPUI	BT	SHLD					PUSH	GS	RSM	BTS	SHRD	IMUL	
	B	CMPXCHG	LSS	BTR	LFS	LGS	MOVZX	POPCB	UD	BT	BTR	BSF	BSR	MOVSB			
	C	XADD	SSE(1,2)				BSWAP										
	D	MMX, SSE(1,2,3)															
	E	MMX, SSE(1,2)															
	F	MMX, SSE(1,2,3)															

Arithmetic & Logic	Prefix
Memory	System & IO
Stack	Multiple Instructions / Extended Instruction Set
Control Flow & Conditional	



Addressing Modes

Prefix	OpCode	ModR	Byte	Displacement	Immediate Data
0000	00	00	00	00000000	00000000
0000	01	00	00	00000000	00000000
0000	02	00	00	00000000	00000000
0000	03	00	00	00000000	00000000
0000	04	00	00	00000000	00000000
0000	05	00	00	00000000	00000000
0000	06	00	00	00000000	00000000
0000	07	00	00	00000000	00000000
0000	08	00	00	00000000	00000000
0000	09	00	00	00000000	00000000
0000	0A	00	00	00000000	00000000
0000	0B	00	00	00000000	00000000
0000	0C	00	00	00000000	00000000
0000	0D	00	00	00000000	00000000
0000	0E	00	00	00000000	00000000
0000	0F	00	00	00000000	00000000
0000	10	00	00	00000000	00000000
0000	11	00	00	00000000	00000000
0000	12	00	00	00000000	00000000
0000	13	00	00	00000000	00000000
0000	14	00	00	00000000	00000000
0000	15	00	00	00000000	00000000
0000	16	00	00	00000000	00000000
0000	17	00	00	00000000	00000000
0000	18	00	00	00000000	00000000
0000	19	00	00	00000000	00000000
0000	1A	00	00	00000000	00000000
0000	1B	00	00	00000000	00000000
0000	1C	00	00	00000000	00000000
0000	1D	00	00	00000000	00000000
0000	1E	00	00	00000000	00000000
0000	1F	00	00	00000000	00000000

SIB Byte Structure

Prefix	OpCode	ModR	Byte	Displacement	Immediate Data
0000	00	00	00	00000000	00000000
0000	01	00	00	00000000	00000000
0000	02	00	00	00000000	00000000
0000	03	00	00	00000000	00000000
0000	04	00	00	00000000	00000000
0000	05	00	00	00000000	00000000
0000	06	00	00	00000000	00000000
0000	07	00	00	00000000	00000000
0000	08	00	00	00000000	00000000
0000	09	00	00	00000000	00000000
0000	0A	00	00	00000000	00000000
0000	0B	00	00	00000000	00000000
0000	0C	00	00	00000000	00000000
0000	0D	00	00	00000000	00000000
0000	0E	00	00	00000000	00000000
0000	0F	00	00	00000000	00000000
0000	10	00	00	00000000	00000000
0000	11	00	00	00000000	00000000
0000	12	00	00	00000000	00000000
0000	13	00	00	00000000	00000000
0000	14	00	00	00000000	00000000
0000	15	00	00	00000000	00000000
0000	16	00	00	00000000	00000000
0000	17	00	00	00000000	00000000
0000	18	00	00	00000000	00000000
0000	19	00	00	00000000	00000000
0000	1A	00	00	00000000	00000000
0000	1B	00	00	00000000	00000000
0000	1C	00	00	00000000	00000000
0000	1D	00	00	00000000	00000000
0000	1E	00	00	00000000	00000000
0000	1F	00	00	00000000	00000000

v1.0 - 30.08.2011  
Contact: Daniel Plohmman - +49 228 73 54 228 - daniel.plohmman@fkie.fraunhofer.de

Source: Intel x86 Instruction Set Reference  
Opcode table presentation inspired by work of Ange Albertini

Source: Daniel Plohmman

# References

- ▶ Chapter 3
- ▶ Intel 64 and IA-32 Architectures Software Developer's Manuals
  - ▶ approx. 6000 pages, be sure to read over weekend, quiz next class :)