x86-32 and x86-64 Assembly (Part 1)

(No one can be told what the Matrix is, you have to see it for yourself)

Emmanuel Fleury
<emmanuel.fleury@u-bordeaux.fr>

LaBRI, Université de Bordeaux, France

October 8, 2019

Overview

1. Assembly Languages
2. Intel x86 CPU Family
3. Intel x86 Architecture
4. Intel x86 Instruction Sets
5. Interruptions & System Calls
6. Assembly In Practice
7. References

Overview

1 Assembly Languages

2 Intel x86 CPU Family

3 Intel x86 Architecture

4 Intel x86 Instruction Sets

5 Interruptions & System Calls

6 Assembly In Practice

7 References

Motivations and Warnings

What is Assembly Good for?

- **Understand the machine** (debugging is easier, less design errors are made, . . .)
- **Better optimization of routines** (manage and tune your compiler options)
- **Code hardware-dependant routines** (compilers, operating systems, . . .)
- **Reverse-engineering and code obfuscation** (malware/driver analysis)

Knowing assembly will enhance your code !

What is Assembly Bad for?

- **Portability is lost** (code is working only for one family of processors)
- **Obfuscate the code** (only a few programmers can read assembly)
- **Debugging is difficult** (most of the debuggers are lost when hitting assembly)
- **Optimization is tedious** (compiler are usually more efficient than humans)

Use it with caution and sparsity !

Unstructured Programming

Assembly is an **unstructured programming language**, meaning that it provides only extremely basic programming control structures such as:

- **Basic expressions** (arithmetic, bitwise and logic operators);
- **Read/write over memory**;
- **Jump operators**;
- **Tests**.

Note that there are **NO**:

- **Procedure call** (argument passing is done manually);
- **Loop facility** (need to use jumps in place);
- **Scope on variables and functions** (everything is global)

Yet, **jumps**, **tests** and **basic read/write** are enough to implement **any program**.

Unstructured Programming (Examples)

Small “Fake” Unstructured Language

- **Expressions**: ‘`expr`’
- **Read Memory**: ‘`(expr)`’
- **Read Variable**: ‘`v`’
- **Label**: ‘`label: instr`’
- **Write to a variable**: ‘`v=expr`’
- **Write to memory**: ‘`(expr)=expr`’
- **Test**: ‘`test expr`’
- **Conditional Jump**: ‘`jmp expr`’

if . . . then . . . else . . .

```
0x0:  test (0x12af) > 0;
0x1:  jmp 0x3;
0x2:  (0x12af) = 0;
0x3:  (0x12af) = (0x12af)-1;
```

Swapping two memory cells

```
0x0:  tmp = (0x12af);
0x1:  (0x12af) = (0x12b4);
0x2:  (0x12b4) = tmp;
```

Computing 2^{10}

```
0x0:  var = 1;
0x1:  i = 10;
0x2: loop:
      var = 2 * var;
0x3:  i = i-1;
0x4:  test i == 0;
0x5:  jmp loop;
```

Unstructured Programming (Exercise)

Fibonacci Sequence

Write an unstructured function computing the Fibonacci sequence til the rank (`n`) that lies at `0xdeadbeef` memory address.

$$\begin{cases} f_0 &= 0, \\ f_1 &= 1, \\ f_n &= f_{n-1} + f_{n-2} \end{cases}$$

Unstructured Programming (Exercise)

Fibonacci Sequence

Write an unstructured function computing the Fibonacci sequence til the rank (`n`) that lies at `0xdeadbeef` memory address.

$$\begin{cases} f_0 &= 0, \\ f_1 &= 1, \\ f_n &= f_{n-1} + f_{n-2} \end{cases}$$

Solution (proposal)

```
0x0: fib = 0;
0x1: f0 = 0;
0x2: f1 = 1;
0x3: n = (0xdeadbeef);
0x4: fib = f1 + f0;
0x5: f0 = f1;
0x6: f1 = fib;
0x7: n = n - 1
0x8: test n > 0;
0x9: jmp 0x4;
```

A Few Words on Assembly

- **Variables**:
  Restricted to a set of registers given by the CPU architecture.
- **Expressions**:
  Restricted to the instructions available on the CPU.
- **Register**:
  Temporary storage unit for intermediate computation.
- **Instruction set**:
  A coherent set of instructions used to encode programs.
- **Opcodes** (Operation Code):
  Instructions are encoded in an hexadecimal format to be more convenient to decode by the machine.
- **Mnemonics**:
  Each opcode is given a "*human readable name*".
- **Operand**:
  Argument of an instruction (they may be several operands for one operation).

Assembly Languages

- **Motorola 68000** (16/32bits architecture, 1979),
- **Accorn ARM** (Advanced RISC Machine) (32/64bits architecture, 1981),
- **MIPS** (Microprocessor without Interlocked Pipeline Stages) (32/64bits architectures, 1981),
- **Intel IA-32** (Intel Architecture) (32bits architecture, 1986),
- **Sun Sparc** (Scalable Processor Architecture) (32/64bits architecture, 1987),
- **Motorola PowerPC** (Performance Optimization With Enhanced RISC Performance Computing) (32/64bits architecture, 1992),
- **DEC Alpha** (64bits architecture, 1992),
- **AMD x86-64** (64bits architecture, 2000)
- **Intel IA-64** (Itanium Intel Architecture) (64bits architecture, 2001).
- ...

Overview

1 Assembly Languages

2 Intel x86 CPU Family

3 Intel x86 Architecture

4 Intel x86 Instruction Sets

5 Interruptions & System Calls

6 Assembly In Practice

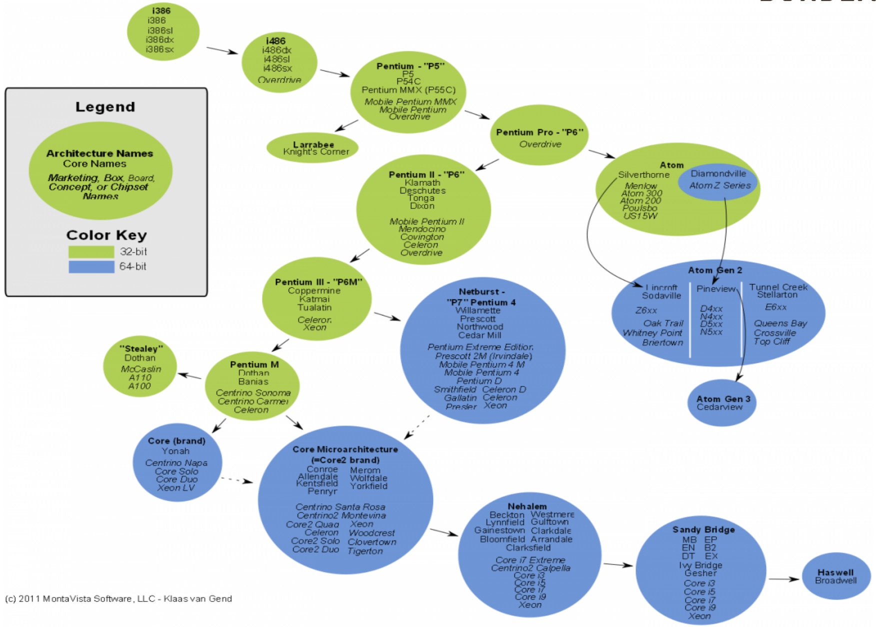
7 References

Early Times

- **Intel 4004** (1971): First microchip ever!
  4bits memory words, 640b of addressable memory, 740kHz
- **Intel 8008** (1972):
  8bits memory words, 16kb of addressable memory, 800kHz
- **Intel 8086** (1978):
  16bits memory words, 1Mb of addressable memory, 10MHz
- **Intel 80286** (1982):
  16bits memory words, 16Mb of addressable memory, 12.5MHz

- **Intel 80386(DX)** (1985): Memory Management Unit (MMU)
  32bits memory words, 4Gb of addressable memory, 16MHz
- **Intel 80486(DX)** (1989): Mathematics co-processor built on-chip
  32bits memory words, 4Gb of addressable memory, 16MHz

Intel CPUs' History



i386 i386 i386sl i386dx i386sx
i486 i486dx i486sl i486sx Overdrive
Pentium - "P5" P5 P54C Pentium MMX (P55C) Mobile Pentium MMX Mobile Pentium Overdrive
Larrabee Knight's Corner
Pentium Pro - "P6" Overdrive
Pentium II - "P6" Klamath Deschutes Tonga Dixon Mobile Pentium II Mendocino Covington Celeron Overdrive
Atom Silverthorne Menlow Atom 300 Atom 200 Poulsbo US15W
Diamondville Atom Z Series
Atom Gen 2 Lincroft Sodaville Z6xx Oak Trail Whitney Point Briertown
Pineview D4xx N4xx D5xx N5xx
Tunnel Creek Stellarton E6xx Queens Bay Crossville Top Cliff
Atom Gen 3 Cedarview
Pentium III - "P6M" Coppermine Katmai Tualatin Celeron Xeon
Netburst - "P7" Pentium 4 Willamette Prescott Northwood Cedar Mill Pentium Extreme Edition Prescott 2M (Irvindale) Mobile Pentium 4 M Mobile Pentium 4 Pentium D Smithfield Celeron D Gallatin Celeron Presler Xeon
"Stealey" Dothan McCaslin A110 A100
Pentium M Dothan Banias Centrino Sonoma Centrino Carmel Celeron
Core (brand) Yonah Centrino Napa Core Solo Core Duo Xeon LV
Core Microarchitecture (=Core2 brand) Conroe Allendale Kentsfield Penryn Merom Wolfdale Yorkfield Centrino Santa Rosa Centrino2 Montevina Core2 Quad Xeon Celeron Woodcrest Core2 Solo Clovertown Core2 Duo Tigerton
Nehalem Beckton Westmere Lynnfield Gulftown Gainestown Clarkdale Bloomfield Arrandale Clarksfield Core i7 Extreme Centrino2 Calpella Core i3 Core i5 Core i7 Core i9 Xeon
Sandy Bridge MB EP EN B2 DT EX Ivy Bridge Gesher Core i3 Core i5 Core i7 Core i9 Xeon
Haswell Broadwell
Legend
Architecture Names
Core Names
Marketing, Box, Board, Concept, or Chipset Names
Color Key
32-bit
64-bit
(c) 2011 MontaVista Software, LLC - Klaas van Gend

Name Soup!

x86-32 Architecture Names

- **AMD**: x86
- **Intel**: IA-32
- **Oracle/Microsoft**: x32
- **GCC**: i386
- **Linux kernel**: x86
- **BSD**: i386
- **Debian/Ubuntu**: i386
- **Fedora/Suse**: i386
- **Gentoo**: x86
- **Solaris**: x86

x86-64 Architecture Names

- **AMD**: x86-64, AMD64
- **Intel**: IA-32e, EM64T, Intel 64
- **Oracle/Microsoft**: x64
- **GCC**: amd64
- **Linux kernel**: x86_64
- **BSD**: amd64
- **Debian/Ubuntu**: amd64
- **Fedora/Suse**: x86_64
- **Gentoo**: amd64
- **Solaris**: amd64

Overview

1 Assembly Languages

2 Intel x86 CPU Family

3 Intel x86 Architecture

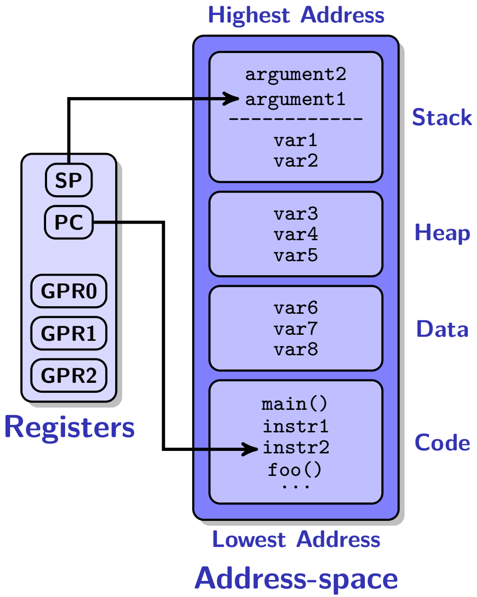
4 Intel x86 Instruction Sets

5 Interruptions & System Calls

6 Assembly In Practice

7 References

Program Overview



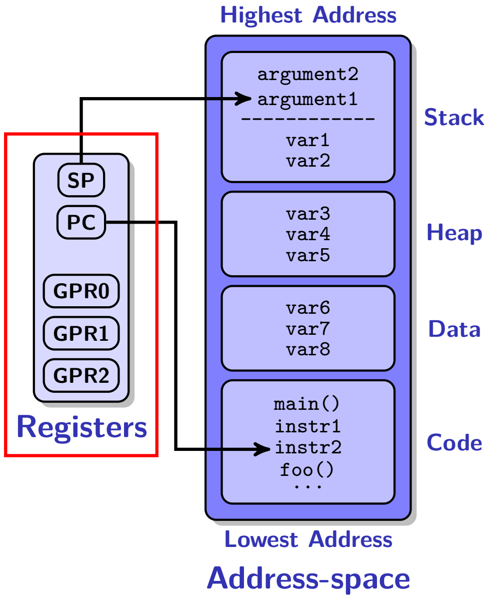
Registers

- SP (Stack Pointer);
- PC (Program Counter);
- GPR (General Purpose Register).

Address-space

- Stack
- Heap
- Data
- Code

Program Overview



Registers

- SP (Stack Pointer);
- PC (Program Counter);
- GPR (General Purpose Register).

Address-space

- Stack
- Heap
- Data
- Code

Overview

1 Assembly Languages

2 Intel x86 CPU Family

3 Intel x86 Architecture
- Intel x86-32 Architecture
- Intel x86-64 Architecture

4 Intel x86 Instruction Sets

5 Interruptions & System Calls

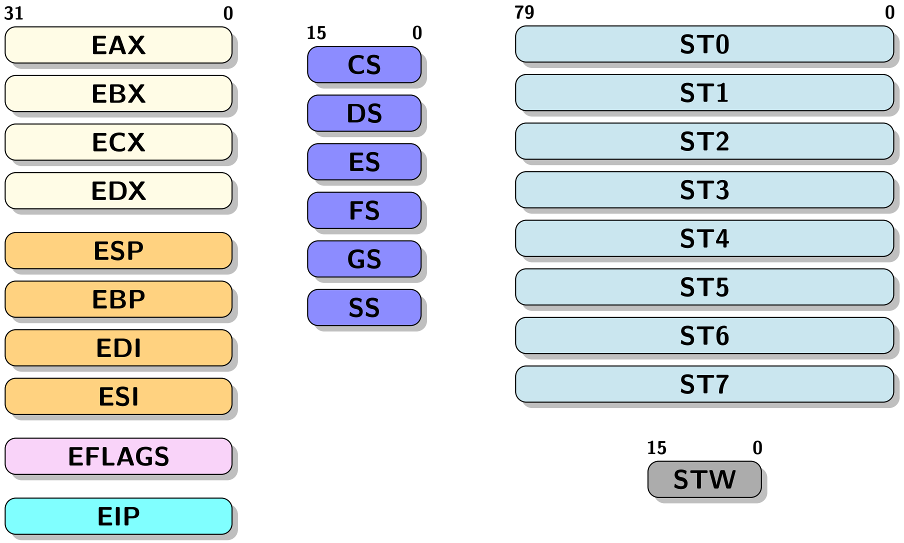
6 Assembly In Practice

7 References

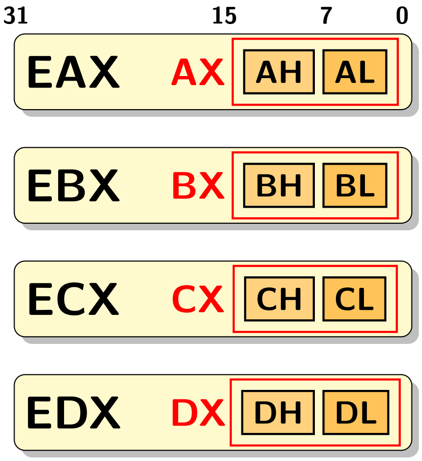
x86-32 Registers

- **Data Registers** (Read/Write)
  (EAX, EBX, ECX, EDX)
- **Index & Pointers Registers** (Read/Write)
  (EBP, ESP, ESI, EDI, EIP)
- **Segment Registers** (Protected)
  (CS, DS, ES, FS, GS, SS)
- **Flags Registers** (Read)
  (EFLAGS)
- **Floating-point Registers** (Read/Write)
  (ST0, . . . , ST7)

x86-32 Registers



x86-32 Data Registers



For **backward compatibility** old **8 bits** and **16 bits** registers have been **preserved**. You still can address them.

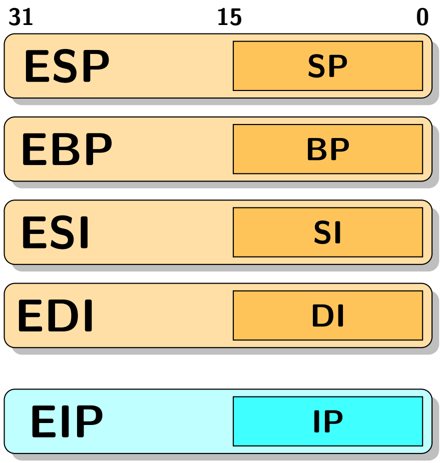
EAX (Accumulator)
Accumulator for operands and results data (addition, subtraction, ...)

EBX (Base Address)
Usually used to store the base address of a data-structure in memory;

ECX (Counter)
Usually used as a loop counter;

EDX (Data Store)
Used to store operand and result for multiplications and divisions.

x86-32 Index & Pointers Registers



For **backward compatibility** old **8 bits** and **16 bits** registers have been **preserved**. You still can address them.

ESP (Stack Pointer)
Pointer to the last cell of the stack;

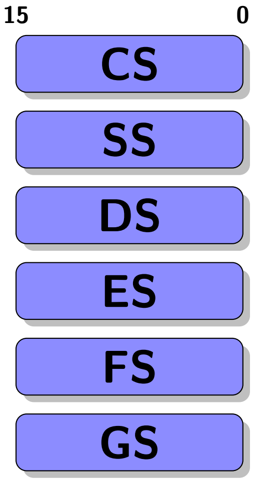
EBP (Base Pointer)
Pointer to the base cell of the current stack frame;

ESI (Source Index)
Used in string operations as source;

EDI (Destination Index)
Used in string operations as destination;

EIP (Instruction Pointer)
Point to the next instruction.

x86-32 Segment Registers



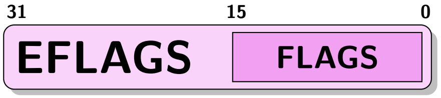
CS (Code Segment)
Point to the current code-segment;

SS (Stack Segment)
Point to the current stack-segment;

DS (Data Segment)
Point to the current data-segment;

ES, **FS**, **GS**, (Extra Data Segments)
Extra segments registers available for far pointer addressing (video memory and others).

x86-32 EFLAGS Register



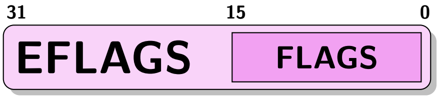
Usage

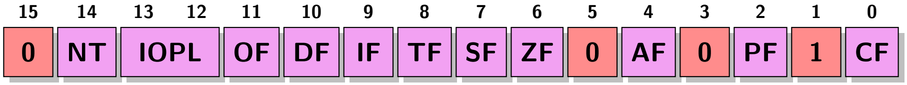
- Most of the instructions modify the EFLAGS to store information about their results.
- Used to store information about tests or arithmetic operators for later usage.
- Used to change the behavior of the CPU or the OS.
- A few bits are reserved and cannot be touched.

Types of flags

- **Status Flags** (stat)
  Result of arithmetic instructions (`add`, `sub`, `mul` or `div`);
- **Control Flags** (ctrl):
  Change the behavior of the processor on some instructions (`std`, `cld`);
- **System Flags** (sys):
  Control system properties (accessible by kernel only).

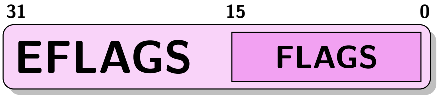
x86-32 Flags Register

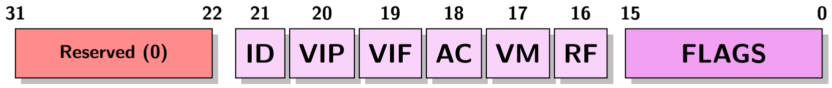




- **CF** (**stat**): Carry flag;
  (left-most bit of the result)
- **PF** (**stat**): Parity flag;
  (right-most bit of the result)
- **AF** (**stat**): Adjust flag;
- **ZF** (**stat**): Zero flag;
  (set if result is zero)
- **SF** (**stat**): Sign flag;
  (most-significant bit of the result)
- **OF** (**stat**): Overflow flag;
  (set if an overflow occurs)
- **DF** (**ctrl**): Direction flag;
  (set the reading direction of a string)
- **IF** (**sys**): Interrupt enabled flag;
- **TF** (**sys**): Trap flag;
- **IOPL** (**sys**): I/O privilege level;
  (ring number currently in use)
- **NT** (**sys**): Nested task flag;

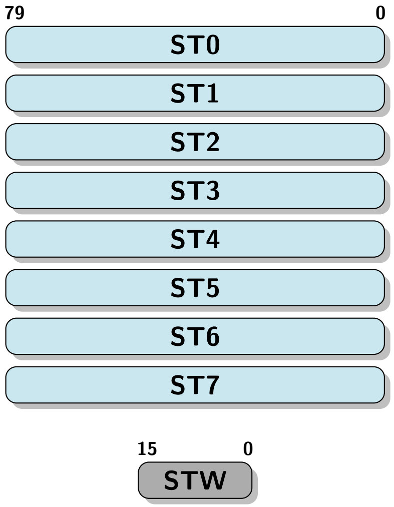
x86-32 Extended Flags Register





- **RF** (**sys**): Resume flag;
  (Set CPU's response to debug exceptions)
- **VM** (**sys**): Virtual 8086 mode;
- **AC** (**sys**): Alignment check;
- **VIF** (**sys**): Virtual interrupt flag;
- **VIP** (**sys**): Virtual interrupt pending;
  (Set if an interrupt is pending)
- **ID** (**sys**): CPUID flag;
  (Ability to use the CPUID instruction)

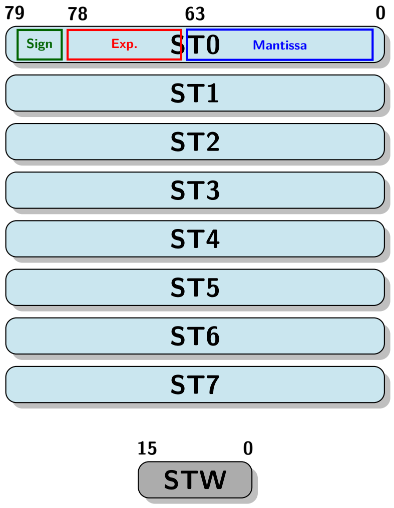
x86-32 Floating-point Registers



FPU Stack Registers

- Floating point numbers are:
  - 32 bits long (C `"float"` type);
  - 64 bits long (C `"double"` type).
- But, to reduce round-off errors, FPU registers are 80 bits wide;
- Registers are accessed as a stack (registers can't be accessed directly)
- Each register contains:
  - Sign: 1 bit;
  - Exponent: 15 bits;
  - Mantissa: 64 bits.

x86-32 Floating-point Registers



FPU Stack Registers

- Floating point numbers are:
  - 32 bits long (C `"float"` type);
  - 64 bits long (C `"double"` type).
- But, to reduce round-off errors, FPU registers are 80 bits wide;
- Registers are accessed as a stack (registers can't be accessed directly)
- Each register contains:
  - Sign: 1 bit;
  - Exponent: 15 bits;
  - Mantissa: 64 bits.

x86-32 Status Word Registers (STW)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	C3	TOP			C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE

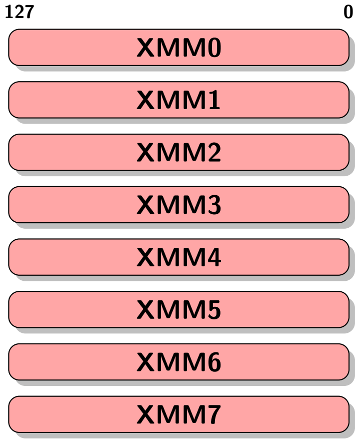
Exception Flags (bits 0-5)

- **IE**: Invalid Operation Exception
- **DE**: Denormalized Operand Exception
- **FE**: Zero Divide Exception
- **OE**: Overflow Exception
- **UE**: Underflow Exception
- **PE**: Precision Exception

Other Flags (bits 6-15)

- **B**: FPU Busy
- **TOP**: Top of Stack Pointer
- **C0**, **C1**, **C2**, **C3**: Condition Code
- **ES**: Error Summary Status
- **SF**: Stack Fault

Streaming SIMD Extensions (SSE Registers)



SSE registers are 8 extra registers 128-bits wide specifically used by SSE instructions.

SSE is an **SIMD instruction set** extension to the x86 architecture added by Intel in the **Pentium III** (1999).

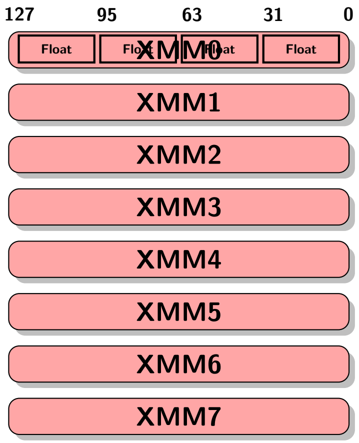
First **SSE** instruction set used only a single data type for XMM registers:

- four 32-bit single-precision floating point numbers.

SSE2 instruction set has later expanded the usage of the XMM registers to include:

- two 64-bit double-precision floating point numbers or,
- two 64-bit integers or,
- four 32-bit integers or,
- eight 16-bit short integers or,
- sixteen 8-bit bytes or characters.

Streaming SIMD Extensions (SSE Registers)



SSE registers are 8 extra registers 128-bits wide specifically used by SSE instructions.

SSE is an **SIMD instruction set** extension to the x86 architecture added by Intel in the **Pentium III** (1999).

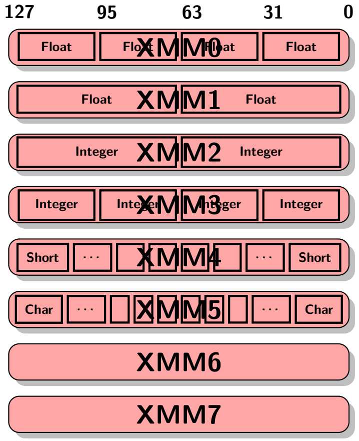
First **SSE** instruction set used only a single data type for XMM registers:

- four 32-bit single-precision floating point numbers.

SSE2 instruction set has later expanded the usage of the XMM registers to include:

- two 64-bit double-precision floating point numbers or,
- two 64-bit integers or,
- four 32-bit integers or,
- eight 16-bit short integers or,
- sixteen 8-bit bytes or characters.

Streaming SIMD Extensions (SSE Registers)



SSE registers are 8 extra registers 128-bits wide specifically used by SSE instructions.

SSE is an **SIMD instruction set** extension to the x86 architecture added by Intel in the **Pentium III** (1999).

First **SSE** instruction set used only a single data type for XMM registers:

- four 32-bit single-precision floating point numbers.

SSE2 instruction set has later expanded the usage of the XMM registers to include:

- two 64-bit double-precision floating point numbers or,
- two 64-bit integers or,
- four 32-bit integers or,
- eight 16-bit short integers or,
- sixteen 8-bit bytes or characters.

Overview

1 Assembly Languages

2 Intel x86 CPU Family

3 **Intel x86 Architecture**
- Intel x86-32 Architecture
- Intel x86-64 Architecture

4 Intel x86 Instruction Sets

5 Interruptions & System Calls

6 Assembly In Practice

7 References

x86-64 Architecture Overview

God/Bad News about x86-64

Good News!

- Instruction are mostly the same;
- More registers to play with:
  – 8 new general purpose 64-bits registers (R8-R15);
  – 8 new SSE 128-bits registers (XMM8-XMM15).

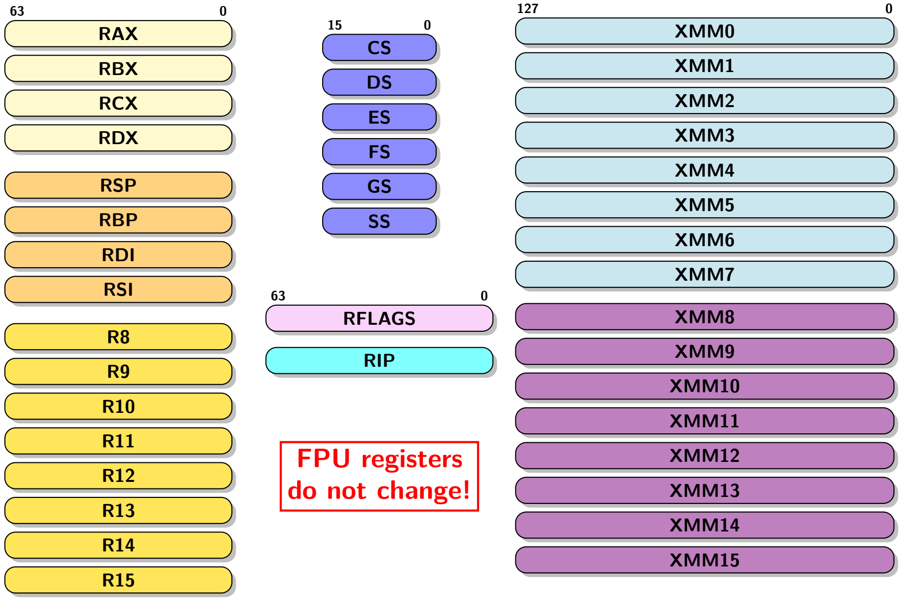
Bad News!

- Calling convention change totally;
- Optimized code is tricky.
  (because of massive usage of SSE instructions)

History of x86-64

- **1999** AMD announces x86-64;
- **2000** AMD releases specs;
- **2001** First x86-64 Linux kernel available;
- **2003** First AMD64 Operton released;
- **2004** Intel announces IA-32e/EM64T, releases first x86-64 Xeon processor;
- **2005** x86-64 versions of Windows XP and Server 2003 released;
- **2009** Mac OS 10.6 (Snow Leopard) includes x86-64 kernel and
  Windows Server 2008 R2 only available in x86-64 version;
- **2010** 50% of Windows 7 installs running the x86-64 version.

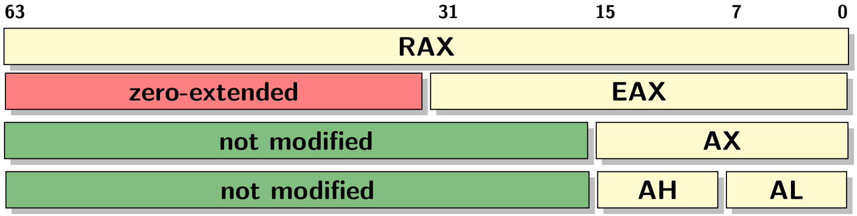
x86-64 Registers



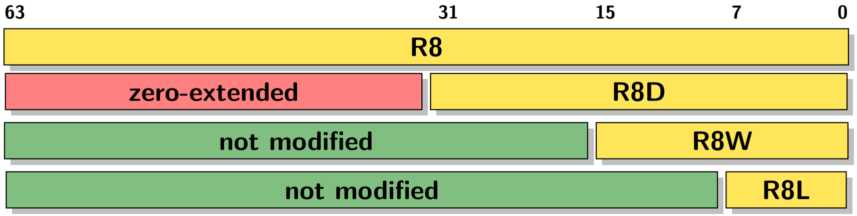
FPU registers do not change!

x86-64 Data Registers

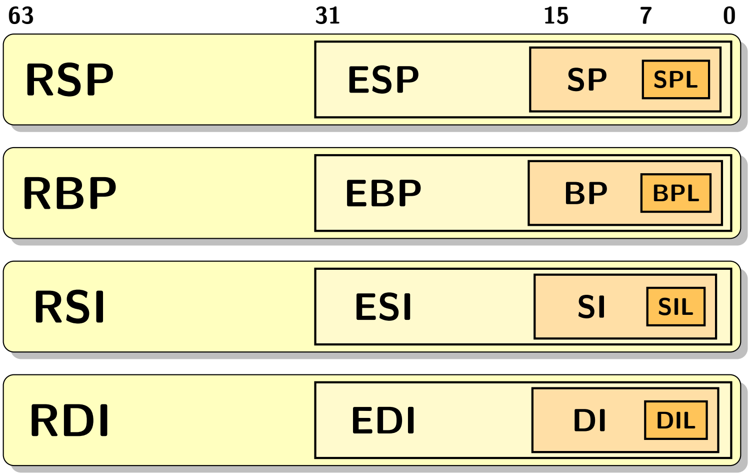
Already Existing registers (RAX-RDX)



New Registers (R8-R15)

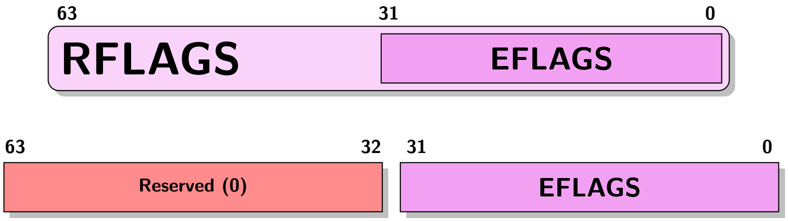


x86-64 Index and Pointers Registers



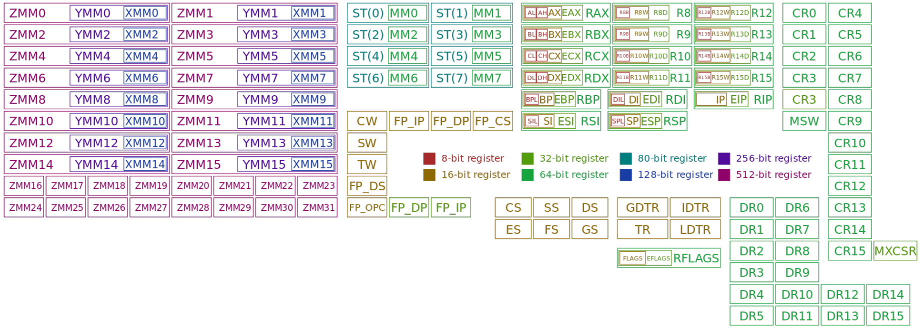
There is a new way to access the low 8-bits chunck of all these registers (SPL, BPL, SIL, DIL).

x86-64 RFLAG Registers



Simply zero-extended (reserved for Intel).
No extra features here (at least to our knowledge).

Real Registers (as we know now!)



Registers available in the x86 instruction set (x86, Wikipedia).

Overview

1 Assembly Languages

2 Intel x86 CPU Family

3 Intel x86 Architecture

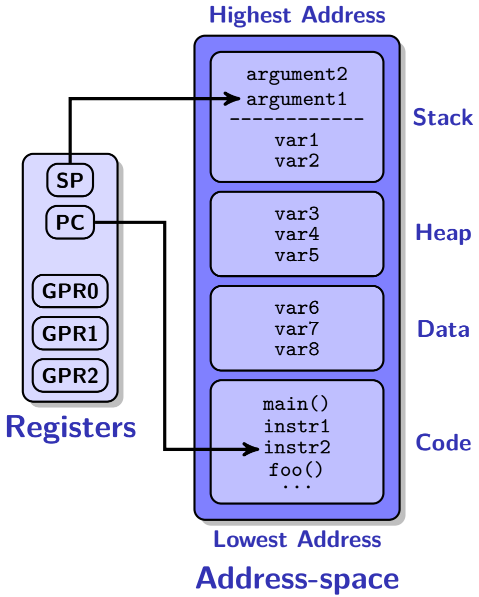
4 **Intel x86 Instruction Sets**

5 Interruptions & System Calls

6 Assembly In Practice

7 References

Program Overview



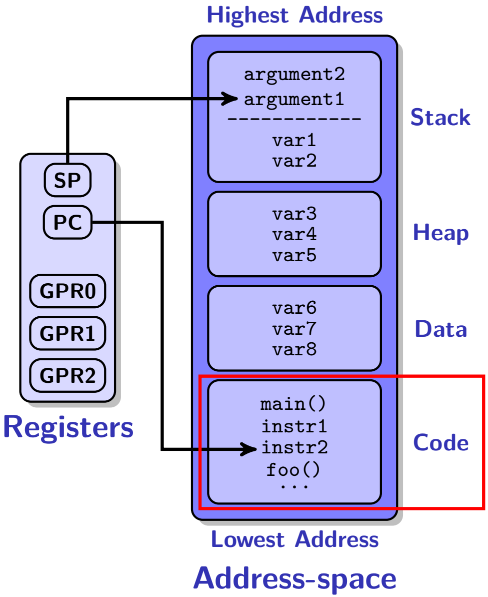
Registers

- SP (Stack Pointer);
- PC (Program Counter);
- GPR (General Purpose Register).

Address-space

- Stack
- Heap
- Data
- Code

Program Overview



Registers

- SP (Stack Pointer);
- PC (Program Counter);
- GPR (General Purpose Register).

Address-space

- Stack
- Heap
- Data
- Code

Syntax and Operational Semantics

(5*3+2)-5 (through registers)

```
movl      $5, %eax # eax = 5
imull     $3, %eax # eax = 3*eax = 15
addl      $2, %eax # eax = eax+2 = 17
subl      $5, %eax # eax = eax-5 = 12
```

Mnemonics

Immediate Values

Registers

If. . . then. . . else. . .

```
_start:
    movl $8, %ebx
    cmpl %eax, %ebx
    jle L0
    incl %ebx
    ret
L0:
    decl %ebx
    ret
```

Small Loop

```
_start:
    movl $8, %eax
    andl $9, %eax
    notl %eax
L0:
    cmp $8, %eax
    jz L0
    ret
```

Syntax and Operational Semantics

(5*3+2)-5 (through registers)

```
→movl       $5, %eax # eax = 5
 imull      $3, %eax # eax = 3*eax = 15
 addl       $2, %eax # eax = eax+2 = 17
 subl       $5, %eax # eax = eax-5 = 12
```

eax $\boxed{5}$

If. . . then. . . else. . .

```
_start:
    movl $8, %ebx
    cmpl %eax, %ebx
    jle L0
    incl %ebx
    ret
L0:
    decl %ebx
    ret
```

Small Loop

```
_start:
    movl $8, %eax
    andl $9, %eax
    notl %eax
L0:
    cmp $8, %eax
    jz L0
    ret
```

Syntax and Operational Semantics

(5*3+2)-5 (through registers)

```
   movl     $5, %eax # eax = 5
→ imull     $3, %eax # eax = 3*eax = 15
   addl     $2, %eax # eax = eax+2 = 17
   subl     $5, %eax # eax = eax-5 = 12
```

eax $\boxed{5}$ *3=15

If. . . then. . . else. . .

```
_start:
    movl $8, %ebx
    cmpl %eax, %ebx
    jle L0
    incl %ebx
    ret
L0:
    decl %ebx
    ret
```

Small Loop

```
_start:
    movl $8, %eax
    andl $9, %eax
    notl %eax
L0:
    cmp $8, %eax
    jz L0
    ret
```

Syntax and Operational Semantics

(5*3+2)-5 (through registers)

```
   movl     $5, %eax # eax = 5
   imull    $3, %eax # eax = 3*eax = 15
 → addl     $2, %eax # eax = eax+2 = 17
   subl     $5, %eax # eax = eax-5 = 12
```

eax $\boxed{15}$ +2=17

If. . . then. . . else. . .

```
_start:
    movl $8, %ebx
    cmpl %eax, %ebx
    jle L0
    incl %ebx
    ret
L0:
    decl %ebx
    ret
```

Small Loop

```
_start:
    movl $8, %eax
    andl $9, %eax
    notl %eax
L0:
    cmp $8, %eax
    jz L0
    ret
```

Syntax and Operational Semantics

(5*3+2)-5 (through registers)

```
   movl      $5, %eax # eax = 5
   imull     $3, %eax # eax = 3*eax = 15
   addl      $2, %eax # eax = eax+2 = 17
→  subl      $5, %eax # eax = eax-5 = 12
```

eax $\boxed{17}$ -5=12

If. . . then. . . else. . .

```
_start:
    movl $8, %ebx
    cmpl %eax, %ebx
    jle L0
    incl %ebx
    ret
L0:
    decl %ebx
    ret
```

Small Loop

```
_start:
    movl $8, %eax
    andl $9, %eax
    notl %eax
L0:
    cmp $8, %eax
    jz L0
    ret
```

Syntax and Operational Semantics

(5*3+2)-5 (through registers)

```
movl      $5, %eax # eax = 5
imull     $3, %eax # eax = 3*eax = 15
addl      $2, %eax # eax = eax+2 = 17
subl      $5, %eax # eax = eax-5 = 12
```

eax $\boxed{12}$

If. . . then. . . else. . .

```
_start:
    movl $8, %ebx
    cmpl %eax, %ebx
    jle L0
    incl %ebx
    ret
L0:
    decl %ebx
    ret
```

Small Loop

```
_start:
    movl $8, %eax
    andl $9, %eax
    notl %eax
L0:
    cmp $8, %eax
    jz L0
    ret
```

AT&T vs. Intel Syntax (1/2)

	AT&T Syntax	Intel Syntax
Community	UNIX	Microsoft

Direction of operands	**from left to right**
	First operand is 'source' and second operand is 'destination'.
	`Instr. src,     dest`
	`mov    (%ecx), %eax`
		from right to left
		First operand is 'destination' and second one is 'source'.
		`Instr. dest, src`
		`mov    eax,  [ecx]`

Addressing Memory	Addresses are enclosed in parenthesis ('`(`', '`)`') and given by the formula: `offset(base, index, scale)`
	`movl (%ebx),                %eax`
	`movl 3(%ebx),               %eax`
	`movl 0x20(%ebx),            %eax`
	`addl (%ebx,%ecx,0x2),       %eax`
	`leal (%ebx,%ecx),           %eax`
	`subl -0x20(%ebx,%ecx,0x4), %eax`
		Addresses are enclosed in brackets ('`[`', '`]`') and given by the formula: `[base+index*scale+offset]`
		`mov eax, [ebx]`
		`mov eax, [ebx+3]`
		`mov eax, [ebx+20h]`
		`add eax, [ebx+ecx*2h]`
		`lea eax, [ebx+ecx]`
		`sub eax, [ebx+ecx*4h-20h]`

AT&T vs. Intel Syntax (2/2)

	AT&T Syntax	Intel Syntax
Data types	● Registers: '%eax' ● Concatenation: '%eax:%ebx' ● Immediate values: '$1' ● Decimal: '10' (or '0d10') ● Hexadecimal: '0x10' ● Operand on bytes: 'movb' ● Operand on words: 'movw' ● Operand on longs: 'movl'	● Registers: 'eax' ● Concatenation: 'eax:ebx' ● Immediate values: '1' ● Decimal: '10' (or '10d') ● Hexadecimal: '10h' ● Address of bytes: 'byte ptr' ● Address of words: 'word ptr' ● Address of longs: 'dword ptr'

```
movl $1,      %eax
movl $0xff,   %ebx
int  $0x80
movb %bl,     %al
movw %bx,     %ax
movl %ebx,    %eax
movl (%ebx),  %eax
```

```
mov eax, 1
mov ebx, 0ffh
int 80h
mov al,  bl
mov ax,  bx
mov eax, ebx
mov eax, dword ptr [ebx]
```

x86 Assembly Instructions

Memory Instructions

Moving data, stack management, strings operators;

Arithmetic & Logic Instructions

Bitwise instructions, shift & rotate, Integer arithmetic;

Control Flow Instructions

Tests, jump, loops;

Floating Point Instructions

FPU stack management, floating-point arithmetic, tests on floating-point values;

Additional Instruction Sets

Other instruction sets (MMX, SSE, . . .).

Moving Data in Memory

Mnemonic	Operand	Operand	Operation
`mov`	src	dst	src $\rightarrow$ dst
`xchg`	src	dst	src $\leftrightarrow$ dst
`lea`	addr	reg	addr $\rightarrow$ reg

Load Effective Address (`lea`)

`lea` calculates its `src` operand as in the `mov` instruction, but rather than loading the contents of that address into `dest`, it loads the address itself.

It can be used for calculating addresses, but also for general-purpose unsigned integer arithmetic (with the caveat and possible benefit that `FLAGS` is untouched).

```
leal  8(,%eax,4), %eax      # Multiply eax by 4 and add 8
leal  (%edx,%eax,2), %eax   # Multiply eax by 2 and add edx
```

Bitwise Instructions

Mnemonic	Operand	Operand	Operation	Touched Flags
`and`	src	dst	src & dst $\rightarrow$ dst	SF,ZF,PF
`or`	src	dst	src | dst $\rightarrow$ dst	SF,ZF,PF
`xor`	src	dst	src ^ dst $\rightarrow$ dst	SF,ZF,PF
`test`	op_1	op_2	op_1 & op_2 (result discarded)	SF,ZF,PF
`not`	dst	–	~dst $\rightarrow$ dst	–
`cmp`	op_1	op_2	op_2 - op_1 (result discarded)	OF,SF,ZF,AF,CF,PF

```
.globl _start

_start:
  movl $8, %eax
  andl $9, %eax
  notl %eax
L0:
  cmp $8, %eax # %eax == 8
  jz L0        # Jump to L0 if ZF==0
```

Shift & Rotate Instructions

Mnemonic	Operand	Operand	Operation	Touched Flags
`shl`	cnt	dst	dst << cnt → dst (unsigned)	CF,OF
`shr`	cnt	dst	dst >> cnt → dst (unsigned)	
`sal`	cnt	dst	dst << cnt → dst (signed)	
`sar`	cnt	dst	dst >> cnt → dst (signed)	
`rol`	cnt	dst	left rotate 'dst' of 'cnt' bits	
`ror`	cnt	dst	right rotate 'dst' of 'cnt' bits	

Multiplication by 2^7

```
.globl _start

_start:
  shll $7, %eax    # %eax * 2 ^ 7
  ret
```

Addition/Subtraction Instructions

Mnemonic	Operand	Operand	Operation	Touched Flags
add	src	dst	src + dst $\rightarrow$ dst	OF,SF,ZF, AF,CF,PF
sub	src	dst	dst - src $\leftrightarrow$ dst	
inc	dst	–	dst + 1 $\rightarrow$ dst	
dec	dst	–	dst - 1 $\rightarrow$ dst	
neg	dst	–	-dst $\rightarrow$ dst	OF,SF,ZF,AF,PF

```
.globl _start

_start:
  movl $15, %eax   # %eax = 15
  subl $7,  %eax   # %eax = %eax - 7
  addl $30, %eax   # %eax = %eax + 30
  decl %eax        # %eax = %eax - 1
```

Multiplication & Division Instructions

Mnemonic	Operand	Operation	Touched Flags
`mul`	src	src × %eax → %eax (unsigned)	CF,OF
`imul`	src	src × %eax → %eax (signed)	
`div`	src	src ÷ %eax → %eax (unsigned)	OF,SF,ZF, AF,CF,PF
`idiv`	src	src ÷ %eax → %eax (signed)	

```
.globl _start

_start:
  movl $8, %eax
  movl $0, %edx
  movl $2, %ebx
  divl %ebx        # %eax = %edx.%eax / %ebx
  addl $3, %eax    # %eax = %eax + 3
  movl $-15, %ebx
  imull %ebx       # %eax = -15 * %eax
```

Jumps (Instructions)

Mnemonic	Operand	Operation
`jmp`	addr	Jump to 'addr' (unconditional jump)
`ja/jge`	addr	Jump to 'addr' if above / greater or equal
`jae/jnb`	addr	Jump to 'addr' if above or equal / not below
`jbe/jna`	addr	Jump to 'addr' if below or equal / not above
`jb/jnae`	addr	Jump to 'addr' if below / not above or equal
`jg/jnle`	addr	Jump to 'addr' if greater / not less or equal
`jge/jnl`	addr	Jump to 'addr' if greater or equal / not less
`jle/jng`	addr	Jump to 'addr' if less or equal / not greater
`jl/jnge`	addr	Jump to 'addr' if less / not greater or equal
`je/jz`	addr	Jump to 'addr' if equal / zero (ZF=1)
`jne/jnz`	addr	Jump to 'addr' if not equal / not zero (ZF=0)
`jc/jnc`	addr	Jump to 'addr' if carry (CF=1) / not carry (CF=0)
`js/jns`	addr	Jump to 'addr' if signed (SF=1) / not signed (SF=0)
...	...	...

Jumps (Example)

```
.globl _start

_start:
  movl $8, %ebx
  cmpl %eax, %ebx  # Compare %eax, %ebx
  jle L0           # If %eax < %ebx go to L0
  incl %ebx        # Increment %ebx (then)
  ret

L0:
  decl %ebx        # Decrement %ebx (else)
  ret
```

FPU Stack Management

Mnemonic	Operand	Operation	Notes
`finit`	–	–	Initialize FPU
`fincstp`	–	$ST+1 \rightarrow ST$	Increase FPU stack pointer
`fdecstp`	–	$ST-1 \rightarrow ST$	Decrease FPU stack pointer
`ffree`	st(i)	$0 \rightarrow st(i)$	Free st(i)
`fldz`	–	$0 \rightarrow st(0)$	Load zero
`fld1`	–	$1 \rightarrow st(0)$	Load one
`fldpi`	–	$\pi \rightarrow st(0)$	Load π
`fld`	addr	$(addr) \rightarrow st(0)$	Load float
	st(i)	$st(i) \rightarrow st(0)$	
`fild`	addr	$(addr) \rightarrow st(0)$	Load integer
`fst`	addr	$st(0) \rightarrow (addr)$	Store float to memory
`fxch`	st(i)	$st(i) \leftrightarrow st(0)$	Exchange content of st(0) and st(i)

FPU Arithmetic

Mnemonic	Operand	Operation	Notes
`fadd`	–	$st(1) + st(0) \rightarrow st(0)$	Float addition
	addr	$(addr) + st(0) \rightarrow st(0)$	
	addr, addr	$(addr) + (addr) \rightarrow st(0)$	
`fsub`	–	$st(1) - st(0) \rightarrow st(0)$	Float subtraction
	addr	$(addr) - st(0) \rightarrow st(0)$	
	addr, addr	$(addr) - (addr) \rightarrow st(0)$	
`fmul`	–	$st(1) \times st(0) \rightarrow st(0)$	Float multiplication
	addr	$(addr) \times st(0) \rightarrow st(0)$	
	addr, addr	$(addr) \times (addr) \rightarrow st(0)$	
`fdiv`	–	$st(1) \div st(0) \rightarrow st(0)$	Float division
	addr	$(addr) \div st(0) \rightarrow st(0)$	
	addr, addr	$(addr) \div (addr) \rightarrow st(0)$	
`fchs`	–	$-st(0) \rightarrow st(0)$	Change sign
`fabs`	–	$\|st(0)\| \rightarrow st(0)$	Absolute value
`fsqr`	–	$st(0)^2 \rightarrow st(0)$	Square
`fsqrt`	–	$\sqrt{st(0)} \rightarrow st(0)$	Square root
`fsin`	–	$\sin(st(0)) \rightarrow st(0)$	Sinus
`fcos`	–	$\cos(st(0)) \rightarrow st(0)$	Cosine

FPU Conditionals

Mnemonic	Operand	Operation	Notes
`fcom`	–	Compares st(0) and st(1)	C0=($st(0) < src$), C3=($st(0) == src$)
	st(i)	Compares st(0) and st(i)	
	addr	Compares st(0) and (addr)	
`fcomi`	st(i)	Compares st(0) and st(i)	Set EFLAGS (not STW)
`fcmovb`	st(i)	if (CF=1) $st(i) \rightarrow st(0)$	Move if below
`fcmove`	st(i)	if (ZF=1) $st(i) \rightarrow st(0)$	Move if equal
`fcmovbe`	st(i)	if (CF=1)&(ZF=1) $st(i) \rightarrow st(0)$	Move if below or equal

FPU Example (#decimal digits)

```
_start:
    movl    $1024, %eax # push the integer (1024) to analyze
    bsrl    %eax, %eax  # bit scan reverse (smallest non zero index)
    inc     %eax        # take the 0th index into account
    pushl   %eax        # save the result on the stack

    fildl   (%esp)      # load to the FPU stack (st(0))
    fldlg2              # load log10(2) on the FPU stack
    fmulp   %st, %st(1) # %st(0) * %st(1) -> %st(0)

    # Set the FPU control word (%stw) to 'round-up' (default: 'round-down')
    fstcw  -2(%esp)      # save the old FPU control word
    movw   -2(%esp), %ax # store the FPU control word in %ax
    andw   $0xf3ff, %ax  # remove everything else
    orw    $0x0800, %ax  # set the 'round-up' bit
    movw   %ax, -4(%esp) # store the value back to the stack
    fldcw  -4(%esp)      # set the FPU control word with the proper value

    frndint              # round-up

    fldcw  -2(%esp)      # restore the old FPU control word

    fistpl (%esp)        # load the final result to the stack
    popl   %eax          # set the return value to be our result

    leave                # clean the stack-frame
```

x86-64 New Instructions

A non-exhaustive list of new x86-64 instructions

Mnemonic	Operand	Operation
`cdqe`	src (addr)	SignExtend(src) $\rightarrow$ `rax`
`cmpsq`	–	Compare(`rsi`,`rdi`)
`cmpxchg16b`	dst (addr)	if (`rdx:rax` != dst) then dst $\rightarrow$ `rdx:rax`
`movsq`	–	(`rsi`) $\rightarrow$ `rdi`
`movzx`	src, dst	ZeroExtend(src) $\rightarrow$ dst
`lodsq`	–	(`rsi`) $\rightarrow$ `rax`
`stosq`	–	(`rax`) $\rightarrow$ (`rdi`)
`syscall`	–	Enter in syscall (replace `sysenter`)
`sysret`	–	Return from syscall (replace `sysexit`)

Other Instruction Sets

Useful Instructions

Mnemonic	Operation
`nop`	No Operation
`hlt`	Stop the CPU until the next interruption occurs

SSE Instructions (Single-precision Floats)

Mnemonic	Operand	Operand	Operation
`movss`	src (addr/xmm)	dst (xmm)	src $\rightarrow$ dst
`addss`	src (addr/xmm)	dst (xmm)	src $+$ dst $\rightarrow$ dst
`subss`	src (addr/xmm)	dst (xmm)	src - dst $\rightarrow$ dst
`mulss`	src (addr/xmm)	dst (xmm)	src $\times$ dst $\rightarrow$ dst
`divss`	src (addr/xmm)	dst (xmm)	src $\div$ dst $\rightarrow$ dst

Overview

1 Assembly Languages

2 Intel x86 CPU Family

3 Intel x86 Architecture

4 Intel x86 Instruction Sets

5 Interruptions & System Calls

6 Assembly In Practice

7 References

What does an interruption do

1. Stop current activity of CPU and save its status;
2. Call a specific subroutine from the OS (interrupt handler);
3. Depending on the interruption call (0-255), the interrupt handler load an interrupt vector which jumps to the corresponding subroutine;
4. If several interruptions are occurring at the same time, CPU has a priority order to apply;
5. When the subroutine is finished the CPU restore the CPU status and continue the previous execution.

Types of Interruptions

Internal Hardware Interrupts

Event occurring during the execution of a program
(division by zero, overflow, general protection error, . . .);

External Hardware Interrupts

Event produced by controllers of external devices
(PCI/AGP bus, hard-drive, graphic cards, keyboard, . . .);

Software Interrupts

Event produced by programs (mainly by the OS).
These interrupts can be produced by using the instruction '`int`'.

List of Interruptions (Linux)

ID	Name
0x00	Division error
0x01	Single-step mode (debug)
0x02	NMI Interrupt
0x03	Breakpoint
0x04	Overflow
0x05	Bound range exceeded
0x06	Invalid opcode
0x07	Coprocessor not available
0x08	Double exception
0x09	Coprocessor segment overrun
0x0a	Invalid TSS (Task State Segment)
0x0b	Segment not present
0x0c	Stack fault
0x0d	General protection
0x0e	Page fault
0x0f	Reserved
0x10	Coprocessor error
0x11 – 0x1f	Error
0x12 – 0xffffff	Undefined

System Calls

A system call is a software interrupt tight to a specific subroutine of the OS (e.g. get a char from keyboard, print on `stdout`, ...).

Calling a syscall

- `int $0x80` (x86-32/x86-64):
  - Syscall ID: `eax`;
  - Syscall arguments: `ebx`, `ecx`, `edx`, `esi`, `edi`.
- `syscall` (x86-64):
  - Syscall ID: `rax`;
  - Syscall arguments: `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`.

Example

```
.globl main
main:
  movl $1, %eax # Interruption ID
  int $0x80     # Calling the OS
  ret
```

A Few System Calls (x86-32)

eax	Name	ebx	ecx	edx	esi	edi
0x01	sys_exit	int	–	–	–	–
0x02	sys_fork	struct pt_regs	–	–	–	–
0x03	sys_read	unsigned int	char*	size_t	–	–
0x04	sys_write	unsigned int	const char*	size_t	–	–
0x05	sys_open	const char*	int	int	–	–
0x06	sys_close	unsigned int	–	–	–	–
0x07	sys_waitpid	pid_t	unsigned int	int	–	–
0x08	sys_create	const char*	int	–	–	–
0x09	sys_link	const char*	const char*	–	–	–
0x0a	sys_unlink	const char*	–	–	–	–
0x0b	sys_execve	const char*	char**	char**	–	–
0x0c	sys_chdir	const char*	–	–	–	–
0x0d	sys_time	int*	–	–	–	–
0x0e	sys_mknod	const char*	mode_t	dev_t	–	–
0x0f	sys_chmod	const char*	mode_t	–	–	–
...	...	...	...	...	...	...

See in `/usr/include/asm/unistd_32.h`

A Few System Calls (x86-64)

rax	Name	rdi	rsi	rdx	r10	r8	r9
0x00	`sys_read`	`unsigned int`	`char*`	`size_t`	–	–	–
0x01	`sys_write`	`unsigned int`	`const char*`	`size_t`	–	–	–
0x02	`sys_open`	`const char*`	`int`	`int`	–	–	–
0x03	`sys_close`	`unsigned int`	–	–	–	–	–
0x04	`sys_stat`	`const char*`	`struct stat*`	–	–	–	–
0x05	`sys_fstat`	`unsigned int`	`struct stat*`	–	–	–	–
0x06	`sys_lstat`	`fconst char*`	`struct stat*`	`int`	–	–	–
0x07	`sys_poll`	`struct poll_fd`	`unsigned int`	`long`	–	–	–
0x08	`sys_lseek`	`unsigned int`	`off_t`	`unsigned int`	–	–	–
...	...	...	...	...	...	...	...

See in `/usr/include/asm/unistd_64.h`

Hello World!

```
.data # Data section
msg:
    .asciz "Hello World!\n" # String
    len = . - msg           # String length

.text          # Text section
.globl main    # Export entry point to ELF linker

main: # Write the string to stdout
  movl $len, %edx # 3rd argument: string length
  movl $msg, %ecx # 2nd argument: pointer to string
  movl $1,   %ebx # 1st argument: file handler (stdout)
  movl $4,   %eax # System call number (sys_write)
  int  $0x80      # Kernel call
  # And exit
  movl $0,   %ebx # 1st argument: exit code
  movl $1,   %eax # System call number (sys_exit)
  int  $0x80      # Kernel call
```

Overview

1. Assembly Languages
2. Intel x86 CPU Family
3. Intel x86 Architecture
4. Intel x86 Instruction Sets
5. Interruptions & System Calls
6. **Assembly In Practice**
7. References

Get the Instruction Sets of your CPU

Use /proc/cpuinfo!

```
#> cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Intel(R) Core(TM)2 Quad CPU Q9505 @ 2.83GHz
stepping        : 10
microcode       : 0xa07
cpu MHz         : 1998.000
cache size      : 3072 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
                  mca cmov pat pse36 clflush dts acpi mmx fxsr sse
                  sse2 ss ht tm pbe syscall nx lm constant_tsc
                  arch_perfmon pebs bts rep_good nopl aperfmperf
                  eagerfpu pni dtes64 monitor ds_cpl vmx smx est tm2
                  ssse3 cx16 xtpr pdcm sse4_1 xsave lahf_lm tpr_shadow
                  vnmi flexpriority dtherm
bugs            :
bogomips        : 5652.91
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

Get the Instruction Sets of your CPU

Use /proc/cpuinfo!

```
#> cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Intel(R) Core(TM)2 Quad CPU Q9505 @ 2.83GHz
stepping        : 10
microcode       : 0xa07
cpu MHz         : 1998.000
cache size      : 3072 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
                  mca cmov pat pse36 clflush dts acpi mmx fxsr sse
                  sse2 ss ht tm pbe syscall nx lm constant_tsc
                  arch_perfmon pebs bts rep_good nopl aperfmperf
                  eagerfpu pni dtes64 monitor ds_cpl vmx smx est tm2
                  ssse3 cx16 xtpr pdcm sse4_1 xsave lahf_lm tpr_shadow
                  vnmi flexpriority dtherm
bugs            :
bogomips        : 5652.91
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

Processor ID

Get the Instruction Sets of your CPU

Use `/proc/cpuinfo`!

```
#> cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Intel(R) Core(TM)2 Quad CPU Q9505 @ 2.83GHz
stepping        : 10
microcode       : 0xa07
cpu MHz         : 1998.000
cache size      : 3072 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
                  mca cmov pat pse36 clflush dts acpi mmx fxsr sse
                  sse2 ss ht tm pbe syscall nx lm constant_tsc
                  arch_perfmon pebs bts rep_good nopl aperfmperf
                  eagerfpu pni dtes64 monitor ds_cpl vmx smx est tm2
                  ssse3 cx16 xtpr pdcm sse4_1 xsave lahf_lm tpr_shadow
                  vnmi flexpriority dtherm
bugs            :
bogomips        : 5652.91
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

Processor ID (processor : 0)

Model Name (model name : Intel(R) Core(TM)2 Quad CPU Q9505 @ 2.83GHz)

Get the Instruction Sets of your CPU

Use /proc/cpuinfo!

```
#> cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Intel(R) Core(TM)2 Quad CPU Q9505 @ 2.83GHz
stepping        : 10
microcode       : 0xa07
cpu MHz         : 1998.000
cache size      : 3072 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
                  mca cmov pat pse36 clflush dts acpi mmx fxsr sse
                  sse2 ss ht tm pbe syscall nx lm constant_tsc
                  arch_perfmon pebs bts rep_good nopl aperfmperf
                  eagerfpu pni dtes64 monitor ds_cpl vmx smx est tm2
                  ssse3 cx16 xtpr pdcm sse4_1 xsave lahf_lm tpr_shadow
                  vnmi flexpriority dtherm
bugs            :
bogomips        : 5652.91
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

Processor ID

Model Name

Processor Frequency

Get the Instruction Sets of your CPU

Use `/proc/cpuinfo`!

```
#> cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Intel(R) Core(TM)2 Quad CPU Q9505 @ 2.83GHz
stepping        : 10
microcode       : 0xa07
cpu MHz         : 1998.000
cache size      : 3072 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
                  mca cmov pat pse36 clflush dts acpi mmx fxsr sse
                  sse2 ss ht tm pbe syscall nx lm constant_tsc
                  arch_perfmon pebs bts rep_good nopl aperfmperf
                  eagerfpu pni dtes64 monitor ds_cpl vmx smx est tm2
                  ssse3 cx16 xtpr pdcm sse4_1 xsave lahf_lm tpr_shadow
                  vnmi flexpriority dtherm
bugs            :
bogomips        : 5652.91
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

Processor ID

Model Name

Processor Frequency

Supported Instruction Sets

Get the Instruction Sets of your CPU

Use /proc/cpuinfo!

```
#> cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Intel(R) Core(TM)2 Quad CPU Q9505 @ 2.83GHz
stepping        : 10
microcode       : 0xa07
cpu MHz         : 1998.000
cache size      : 3072 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
                  mca cmov pat pse36 clflush dts acpi mmx fxsr sse
                  sse2 ss ht tm pbe syscall nx lm constant_tsc
                  arch_perfmon pebs bts rep_good nopl aperfmperf
                  eagerfpu pni dtes64 monitor ds_cpl vmx smx est tm2
                  ssse3 cx16 xtpr pdcm sse4_1 xsave lahf_lm tpr_shadow
                  vnmi flexpriority dtherm
bugs            :
bogomips        : 5652.91
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

Processor ID (processor)

Model Name (model name)

Processor Frequency (cpu MHz)

Supported Instruction Sets (flags)

- `fpu`: Floating point unit
- `ht`: Hyper-threads
- `syscall`: System call instructions
- `nx`: Non-executable memory pages
- `lm`: Long mode (64-bits extensions)
- `vmx/svm`: Hardware virtualization (Intel/AMD)

Compiling Assembly with GCC

With libc

```
# Example with libc
.globl main

main:
  movl $20, %eax
  ret
```

Build the binary

```
gcc -m32 -static -o ex1 asm.s
gcc -m64 -static -o ex1 asm.s
```

Run the binary

```
#> ./ex1
```

Without libc

```
# Example with no libc
.globl _start

_start:
  movl $20, %eax
  # No 'ret' in _start!!!
```

Build the binary

```
gcc -m32 -static -nostdlib -o ex1 asm.s
gcc -m64 -static -nostdlib -o ex1 asm.s
```

Run the binary

```
#> ./ex1
```

Compiling Assembly with `as/ld`

Witout `libc`

```
# Example with no libc
.globl _start

_start:
  movl $20, %eax
  # No 'ret' in _start!!!
```

Build the binary (32)

```
as --32 -o ex1.o asm.s
ld -m elf_i386 -o ex1 ex1.o
```

Run the binary

```
#> ./ex1
```

Build the binary (64)

```
as --64 -o ex1.o asm.s
ld -m elf_x86_64 -o ex1 ex1.o
```

Run the binary

```
#> ./ex1
```

Using GDB with Assembly

```
#> gdb ./ite
...
Reading symbols from ./ite...done.
(gdb) break _start
Breakpoint 1 at 0x175: file ite.s, line 5.
(gdb) run
Starting program: ./ite

Breakpoint 1, _start () at ite.s:5
5           movl $8, %ebx
(gdb) disas
Dump of assembler code for function _start:
=> 0x56555175 <+0>:        mov    $0x8,%ebx
   0x5655517a <+5>:     cmp    %eax,%ebx
   0x5655517c <+7>:     jle    0x56555180 <L0>
   0x5655517e <+9>:     inc    %ebx
   0x5655517f <+10>:    ret
End of assembler dump.
(gdb) disas L0
Dump of assembler code for function L0:
   0x56555180 <+0>:     dec    %ebx
   0x56555181 <+1>:     ret
End of assembler dump.
(gdb) nexti
6           cmpl %eax, %ebx
(gdb) stepi
7           jle L0
(gdb) si
8           incl %ebx
(gdb) disas
Dump of assembler code for function _start:
   0x56555175 <+0>:     mov    $0x8,%ebx
   0x5655517a <+5>:     cmp    %eax,%ebx
   0x5655517c <+7>:     jle    0x56555180 <L0>
=> 0x5655517e <+9>:     inc    %ebx
   0x5655517f <+10>:    ret
End of assembler dump.
```

```
(gdb) backtrace
#0  _start () at ite.s:8
(gdb)  print /x $eax
$1 = 0xf7ffd918
(gdb) set $eax = 0
(gdb) info reg
eax            0x0              0
ecx            0xffffd134       -11980
edx            0xf7fe88b0       -134313808
ebx            0x8              8
esp            0xffffd130       0xffffd130
ebp            0x0              0x0
esi            0xffffd13c       -11972
edi            0x56555175       1448431989
eip            0x5655517e       0x5655517e <_start+9>
eflags         0x207            [ CF PF IF ]
...
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: ./ite

Breakpoint 1, _start () at ite.s:5
5           movl $8, %ebx
(gdb) set $eax = 10
(gdb) si
6           cmpl %eax, %ebx
(gdb) si
7           jle L0
(gdb) si
11          decl %ebx
(gdb) disas
Dump of assembler code for function L0:
=> 0x56555180 <+0>:     dec    %ebx
   0x56555181 <+1>:     ret
End of assembler dump.
(gdb)
```

Using objdump

```
#> objdump -d ite

ite: file format elf32-i386
Disassembly of section .text:

00000175 <_start>:
 175:   bb 08 00 00 00       mov    $0x8,%ebx
 17a:   39 c3                cmp    %eax,%ebx
 17c:   7e 02                jle    180 <L0>
 17e:   43                   inc    %ebx
 17f:   c3                   ret

00000180 <L0>:
 180:   4b                   dec    %ebx
 181:   c3                   ret
```

Using objdump

Instruction addresses

```
#> objdump -d ite

ite: file format elf32-i386
Disassembly of section .text:

00000175 <_start>:
 175:   bb 08 00 00 00        mov    $0x8,%ebx
 17a:   39 c3                 cmp    %eax,%ebx
 17c:   7e 02                 jle    180 <L0>
 17e:   43                    inc    %ebx
 17f:   c3                    ret

00000180 <L0>:
 180:   4b                    dec    %ebx
 181:   c3                    ret
```

Using objdump

Instruction addresses

Instruction opcodes

```
#> objdump -d ite

ite: file format elf32-i386
Disassembly of section .text:

00000175 <_start>:
 175:   bb 08 00 00 00       mov    $0x8,%ebx
 17a:   39 c3                cmp    %eax,%ebx
 17c:   7e 02                jle    180 <L0>
 17e:   43                   inc    %ebx
 17f:   c3                   ret

00000180 <L0>:
 180:   4b                   dec    %ebx
 181:   c3                   ret
```

Using objdump

Instruction addresses

Instruction opcodes

Instruction mnemonics

```
#> objdump -d ite

ite: file format elf32-i386
Disassembly of section .text:

00000175 <_start>:
 175:   bb 08 00 00 00          mov    $0x8,%ebx
 17a:   39 c3                   cmp    %eax,%ebx
 17c:   7e 02                   jle    180 <L0>
 17e:   43                      inc    %ebx
 17f:   c3                      ret

00000180 <L0>:
 180:   4b                      dec    %ebx
 181:   c3                      ret
```

Using `objdump`

Instruction addresses

Instruction opcodes

Instruction mnemonics

Instruction operands

```
#> objdump -d ite

ite: file format elf32-i386
Disassembly of section .text:

00000175 <_start>:
 175:   bb 08 00 00 00        mov    $0x8,%ebx
 17a:   39 c3                 cmp    %eax,%ebx
 17c:   7e 02                 jle    180 <L0>
 17e:   43                    inc    %ebx
 17f:   c3                    ret

00000180 <L0>:
 180:   4b                    dec    %ebx
 181:   c3                    ret
```

Using objdump

```
#> objdump -d ite

ite: file format elf32-i386
Disassembly of section .text:

00000175 <_start>:
 175:    bb 08 00 00 00         mov     $0x8,%ebx
 17a:    39 c3                  cmp     %eax,%ebx
 17c:    7e 02                  jle     180 <L0>
 17e:    43                     inc     %ebx
 17f:    c3                     ret

00000180 <L0>:
 180:    4b                     dec     %ebx
 181:    c3                     ret
```

Symbols

Using radare2

```
#> r2 ./ite

[0x00000175]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))

[0x00000175]> pdf
            ;-- section_end..dynstr:
            ;-- section..text:
            ;-- _start:
/ (fcn) entry0 13
|   entry0 ();
|           0x00000175  bb08000000  mov ebx, 8   ; [6] va=0x175 sz=13 rwx=--r-x .text
|           0x0000017a  39c3        cmp ebx, eax
|       ,=< 0x0000017c  7e02        jle loc.L0
|       |   0x0000017e  43          inc ebx
|       |   0x0000017f  c3          ret
|       `-> ;-- L0:
|       |   ; JMP XREF from 0x0000017c (entry0)
|       `-> 0x00000180  4b          dec ebx
\           0x00000181  c3          ret
[0x00000175]>
```

Using radare2

```
#> r2 ./ite

[0x00000175]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))

[0x00000175]> pdf
            ;-- section_end..dynstr:
            ;-- section..text:
            ;-- _start:
/ (fcn) entry0 13
|   entry0 ();
|           0x00000175  bb08000000  mov ebx, 8   ; [6] va=0x175 sz=13 rwx=--r-x .text
|           0x0000017a  39c3        cmp ebx, eax
|       ,=< 0x0000017c  7e02        jle loc.L0
|       |   0x0000017e  43          inc ebx
|       |   0x0000017f  c3          ret
|       `-> ;-- L0:
|       |   ; JMP XREF from 0x0000017c (entry0)
|       `-> 0x00000180  4b          dec ebx
\           0x00000181  c3          ret
[0x00000175]>
```

pdf: Print Disassembly Functions

The principle of Radare2 is to use one-letter commands that can be combined into more advanced commands. For example, 'pd' (Print Disassembly) is a familly of commands among which can be found the 'pdf' command. To get the list of all commands of the family, just do: 'pd?'

Overview

1 Assembly Languages

2 Intel x86 CPU Family

3 Intel x86 Architecture

4 Intel x86 Instruction Sets

5 Interruptions & System Calls

6 Assembly In Practice

7 References

References I

Intel Corporation.
Intel 64 and IA-32 Architectures Optimization Reference Manual, April 2012.

Intel Corporation.
Intel 64 and IA-32 Architectures Software Developer's Manual, August 2012.

Randall Hyde.
Write Great Code: Understanding the Machine, volume 1.
NoStarch Press, 2004.

Randall Hyde.
Write Great Code: Thinking Low-Level, Writing High-Level, volume 2.
NoStarch Press, 2006.

References II

- Randall Hyde.
  The Art of Assembly Language.
  Number ISBN: 978-1-59327-207-4. NoStarch Press, second edition, 2010.

- Jon Larimer.
  Intro to x64 reversing.
  Talk at SummerCon'2011, NYC, USA, 2011.

Next Time...

x86-32 and x86-64 Assembly (Part 2)