MACHINE-LEVEL PROGRAMMING I: BASICS

CS 429H: SYSTEMS I

Instructor: Emmett Witchel
Today: Machine Programming I: Basics

• History of Intel processors and architectures
• C, assembly, machine code
• Assembly Basics: Registers, operands, move
Intel x86 Processors

- Totally 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td>• First 16-bit processor. Basis for IBM PC &amp; DOS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• 1MB address space</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td>• First 32 bit processor, referred to as IA32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Added “flat addressing”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Capable of running Unix</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• 32-bit Linux/gcc uses no instructions introduced in later models</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>• First 64-bit processor, referred to as x86-64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
</tbody>
</table>

# Intel x86 Processors: Overview

<table>
<thead>
<tr>
<th>Architectures</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>X86-16</td>
<td>8086</td>
</tr>
<tr>
<td>X86-32/IA32</td>
<td>286</td>
</tr>
<tr>
<td>X86-64 / EM64t</td>
<td>386, 486, Pentium,</td>
</tr>
<tr>
<td></td>
<td>Pentium MMX</td>
</tr>
<tr>
<td>MMX</td>
<td>Pentium III</td>
</tr>
<tr>
<td>SSE</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>SSE2</td>
<td>Pentium 4E</td>
</tr>
<tr>
<td>SSE3</td>
<td>Pentium 4F</td>
</tr>
<tr>
<td>SSE4</td>
<td>Core 2 Duo, Core i7</td>
</tr>
</tbody>
</table>

IA: often redefined as latest Intel architecture
Intel x86 Processors, contd.

• Machine Evolution
  • 386 1985
  • Pentium 1993
  • Pentium/MMX 1997
  • PentiumPro 1995
  • Pentium III 1999
  • Pentium 4 2001
  • Core 2 Duo 2006
  • Core i7 2008

• Added Features
  • Instructions to support multimedia operations
    • Parallel operations on 1, 2, and 4-byte data, both integer & FP
  • Instructions to enable more efficient conditional operations

• Linux/GCC Evolution
  • Two major steps: 1) support 32-bit 386.  2) support 64-bit x86-64
x86 Clones: Advanced Micro Devices (AMD)

• Historically
  • AMD has followed just behind Intel
  • A little bit slower, a lot cheaper

• Then
  • Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  • Built Opteron: tough competitor to Pentium 4
  • Developed x86-64, their own extension to 64 bits
Intel’s 64-Bit

- Intel Attempted Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- AMD Stepped in with Evolutionary Solution
  - x86-64 (now called “AMD64”)
- Intel Felt Obligated to Focus on IA64
  - Hard to admit mistake or that AMD is better
- 2004: Intel Announces EM64T extension to IA32
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- All but low-end x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode
Our Coverage

- **IA32**
  - The traditional x86

- **x86-64/EM64T**
  - The emerging standard

- **Presentation**
  - Book presents IA32 in Sections 3.1—3.12
  - Covers x86-64 in 3.13
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
Definitions

- **Architecture**: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
  - Examples: instruction set specification, registers.

- **Microarchitecture**: Implementation of the architecture.
  - Examples: cache sizes and core frequency.

- Example ISAs (Intel): x86, IA, IPF
Assembly Programmer's View

- **Programmer-Visible State**
  - PC: Program counter
    - Address of next instruction
    - Called “EIP” (IA32) or “RIP” (x86-64)
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures
Program to Process

• We write a program in e.g., C.
• A compiler turns that program into an instruction list.
• The CPU interprets the instruction list (which is more a graph of basic blocks).

```c
void X (int b) {
    if(b == 1) {
        ...
        int main() {
            int a = 2;
            X(a);
        }
    }
}
```
# Process in Memory

- Program to process.
  - **What you wrote**
    ```c
    void X (int b) {
        if(b == 1) {
            ...
        }
        int main() {
            int a = 2;
            X(a);
        }
    }
    ```
  - **What must the OS track for a process?**

- **What is in memory.**

<table>
<thead>
<tr>
<th>main; a = 2</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>X; b = 2</td>
<td></td>
</tr>
</tbody>
</table>

![Heap Diagram](image)

```c
void X (int b) {
    if(b == 1) {
        ...
    }
    int main() {
        int a = 2;
        X(a);
    }
}
```
A shell forks and execs a calculator

```c
int pid = fork();
if (pid == 0) {
    close(".history");
    exec("/bin/calc");
} else {
    wait(pid);
}
```

```c
int pid = fork();
if (pid == 0) {
    close(".history");
    exec("/bin/calc");
} else {
    wait(pid);
}
```
A shell forks and then execs a calculator

```c
int shell_main() {
    int a = 2;
    ...
}
```

```c
int calc_main() {
    int q = 7;
    ...
}
```

pid = 127
open files = ".history"
last_cpu = 0

pid = 128
open files = ".history"
last_cpu = 0
Anatomy of a Process

Executable File

- Header
- Code
- Initialized data

Process’s address space

- mapped segments
  - DLL’s
  - Stack
  - Heap
  - Initialized data
  - Code
Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

```
text
C program (p1.c p2.c)
```

```
text
Asm program (p1.s p2.s)
```

```
binary
Object program (p1.o p2.o)
```

```
binary
Executable program (p)
```

Static libraries (.a)

Compiler (gcc -S)

Assembler (gcc or as)

Linker (gcc or ld)
Compiling Into Assembly

C 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
    popl %ebp
    ret
```

Some compilers use instruction “leave”

Obtain with command

```
/usr/local/bin/gcc -O1 -S code.c
```

Produces file code.s
Assembly Characteristics: Data Types

• “Integer” data of 1, 2, or 4 bytes
  • Data values
  • Addresses (untyped pointers)

• Floating point data of 4, 8, or 10 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
Object Code

Code for `sum`

0x401040 <sum>:

- 0x55
- 0x89
- 0xe5
- 0x8b
- 0x0c
- 0x03
- 0x45
- 0x08
- 0x5d
- 0xc3

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

- **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

---

Object 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
Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>080483c4</td>
<td>&lt;sum&gt;</td>
<td>55 push %ebp</td>
<td></td>
</tr>
<tr>
<td>080483c5</td>
<td></td>
<td>89 e5 mov %esp,%ebp</td>
<td></td>
</tr>
<tr>
<td>080483c7</td>
<td></td>
<td>8b 45 0c mov 0xc(%ebp),%eax</td>
<td></td>
</tr>
<tr>
<td>080483ca</td>
<td></td>
<td>03 45 08 add 0x8(%ebp),%eax</td>
<td></td>
</tr>
<tr>
<td>080483cd</td>
<td></td>
<td>5d pop %ebp</td>
<td></td>
</tr>
<tr>
<td>080483ce</td>
<td></td>
<td>c3 ret</td>
<td></td>
</tr>
</tbody>
</table>

- **Disassembler**

  `objdump -d p`

  - 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

Disassembled

Dump of assembler code for function sum:
0x080483c4 <sum+0>: push %ebp
0x080483c5 <sum+1>: mov %esp,%ebp
0x080483c7 <sum+3>: mov 0xc(%ebp),%eax
0x080483ca <sum+6>: add 0x8(%ebp),%eax
0x080483cd <sum+9>: pop %ebp
0x080483ce <sum+10>: ret

• Within gdb Debugger
  
gdb p
disassemble sum
  • Disassemble procedure
  
x/11xb sum
  • Examine the 11 bytes starting at sum
What Can be Disassembled?

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

% 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
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
Integer Registers (IA32)

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

- %ax
- %cx
- %dx
- %bx
- %si
- %di
- %sp
- %bp

- %ah
- %ch
- %dh
- %bh
- %al
- %cl
- %dl
- %bl

General purpose registers

16-bit virtual registers (backwards compatibility)

Origin
(mostly obsolete)

- accumulate
- counter
- data
- base
- source
- index
- destination
- index
- stack
- pointer
- base
- pointer
Simple Memory Addressing Modes

- **Normal (R)** \( \text{Mem}[\text{Reg}[R]] \)
  - Register \( R \) specifies memory address

  \[
  \text{movl} \ (%\text{ecx}),\%\text{eax} 
  \]

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

  \[
  \text{movl} \ 8(\%\text{ebp}),\%\text{edx} 
  \]
Using Simple Addressing Modes

### void swap(int *xp, int *yp)

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

### Body

```assembly
set up
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)

finish
popl %ebx
popl %ebp
ret
```
Using Simple Addressing Modes

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

```
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
```
## Understanding Swap

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

### Stack (in memory)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
</tr>
<tr>
<td>12</td>
<td>yp</td>
</tr>
</tbody>
</table>

### Register Values

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>xp</td>
</tr>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
</tr>
</tbody>
</table>

### Machine Code

```
movl  8(%ebp), %edx  # edx = xp
movl  12(%ebp), %ecx # ecx = yp
movl  (%edx), %ebx   # ebx = *xp (t0)
movl  (%ecx), %eax   # eax = *yp (t1)
movl  %eax, (%edx)   # *xp = t1
movl  %ebx, (%ecx)   # *yp = t0
```
Understanding Swap

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x104</td>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>0x120</td>
<td>0x11c</td>
</tr>
<tr>
<td>0x120</td>
<td>0x114</td>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
<td>0x114</td>
<td>0x118</td>
</tr>
<tr>
<td>0x10c</td>
<td>0x108</td>
<td>0x104</td>
</tr>
<tr>
<td>0x104</td>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

```
movl  8(%ebp), %edx  # edx = xp
movl  12(%ebp), %ecx # ecx = yp
movl  (%edx), %ebx   # ebx = *xp (t0)
movl  (%ecx), %eax   # eax = *yp (t1)
movl  %eax, (%edx)   # *xp = t1
movl  %ebx, (%ecx)   # *yp = t0
```
## Understanding Swap

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x124</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Code Snippet:

```assembly
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx    # ebx = *xp (t0)
movl (%ecx), %eax    # eax = *yp (t1)
movl %eax, (%edx)    # *xp = t1
movl %ebx, (%ecx)    # *yp = t0
```
Understanding Swap

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>0x110</td>
</tr>
<tr>
<td>0</td>
<td>0x108</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x114</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x118</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x124</td>
</tr>
<tr>
<td>0x124</td>
<td>456</td>
</tr>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
</tbody>
</table>

```
movl 8(%%ebp), %%edx    # edx = xp
movl 12(%%ebp), %%ecx   # ecx = yp
movl (%edx), %ebx       # ebx = *xp (t0)
movl (%ecx), %eax       # eax = *yp (t1)
movl %eax, (%edx)       # *xp = t1
movl %ebx, (%ecx)       # *yp = t0
```
### Understanding Swap

#### Register Values

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

#### Offset Table

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
</tbody>
</table>

#### Instructions

- `movl 8(%ebp), %edx` \# edx = xp
- `movl 12(%ebp), %ecx` \# ecx = yp
- `movl (%edx), %ebx` \# ebx = *xp (t0)
- `movl (%ecx), %eax` \# eax = *yp (t1)
- `movl %eax, (%edx)` \# *xp = t1
- `movl %ebx, (%ecx)` \# *yp = t0
Understanding Swap

| %eax  | 456 |
| %edx  | 0x124 |
| %ecx  | 0x120 |
| %ebx  | 123 |
| %esi  |       |
| %edi  |       |
| %esp  |       |
| %ebp  | 0x104 |

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12 0x120</td>
</tr>
<tr>
<td>xp</td>
<td>8 0x124</td>
</tr>
<tr>
<td></td>
<td>4 0x10c</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
Understanding Swap

\[
\begin{array}{|c|c|}
\hline
\%eax & 456 \\
\%edx & 0x124 \\
\%ecx & 0x120 \\
\%ebx & 123 \\
\%esi & \\
\%edi & \\
\%esp & \\
\%ebp & 0x104 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Address} & 0x124 \\
 & 0x120 \\
 & 0x11c \\
 & 0x118 \\
 & 0x114 \\
 & 0x110 \\
 & 0x10c \\
 & 0x108 \\
 & 0x104 \\
 & 0x100 \\
\hline
\text{Offset} & \\
yp & 12 \\
xp & 8 \\
 & 4 \\
\%ebp & 0 \\
 & -4 \\
\hline
\end{array}
\]

\[
\begin{align*}
\text{movl} & \ 8(\%ebp), \ %edx \quad \# \ edx = xp \\
\text{movl} & \ 12(\%ebp), \ %ecx \quad \# \ ecx = yp \\
\text{movl} & \ (%edx), \ %ebx \quad \# \ ebx = *xp \ (t0) \\
\text{movl} & \ (%ecx), \ %eax \quad \# \ eax = *yp \ (t1) \\
\text{movl} & \ %eax, \ (%edx) \quad \# \ *xp = t1 \\
\text{movl} & \ %ebx, \ (%ecx) \quad \# \ *yp = t0 \\
\end{align*}
\]
Understanding Swap

| %eax | 456 |
| %edx | 0x124 |
| %ecx | 0x120 |
| %ebx | 123 |
| %esi | |
| %edi | |
| %esp | |
| %ebp | 0x104 |

```
movl 8(%ebp), %edx    # edx = xp
movl 12(%ebp), %ecx   # ecx = yp
movl (%edx), %ebx     # ebx = *xp (t0)
movl (%ecx), %eax     # eax = *yp (t1)
movl %eax, (%edx)     # *xp = t1
movl %ebx, (%ecx)     # *yp = t0
```
Complete Memory Addressing Modes

• Most General Form

\[ D(Rb, Ri, S) \quad Mem[Reg[Rb] + S*Reg[Ri] + D] \]

• D: Constant “displacement” 1, 2, or 4 bytes
• Rb: Base register: Any of 8 integer registers
• Ri: Index register: Any, except for %esp
  • Unlikely you’d use %ebp, either
• S: Scale: 1, 2, 4, or 8 (why these numbers?)

• Special Cases

(Rb, Ri) \quad Mem[Reg[Rb] + Reg[Ri]]
D(Rb, Ri) \quad Mem[Reg[Rb] + Reg[Ri] + D]
(Rb, Ri, S) \quad Mem[Reg[Rb] + S*Reg[Ri]]
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Extend existing registers. Add 8 new ones.
- Make `%ebp/%rbp` general purpose