# Machine-Level Programming: Introduction

CMPU 224 – Computer Organization
Jason Waterman

# Intel x86 Processors

- Dominate laptop/desktop/server market

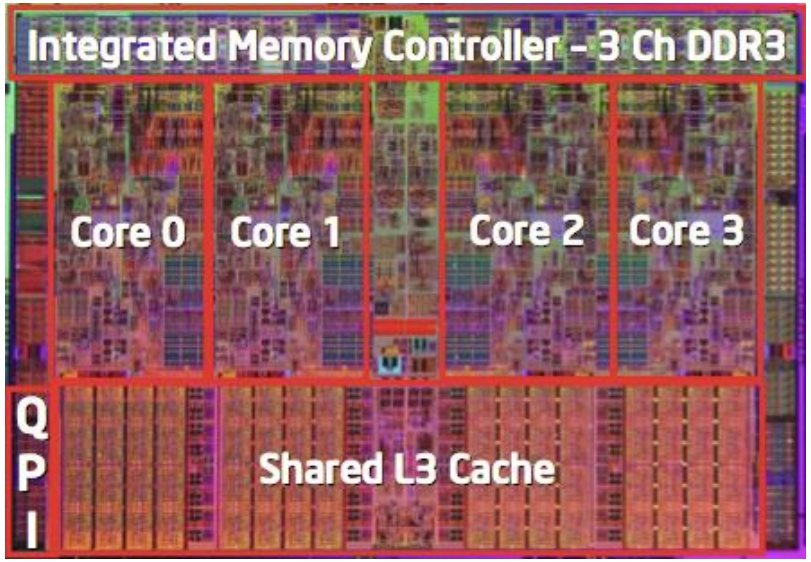
- Evolutionary design
  - Backwards compatible with the 8086, introduced in 1978
  - Added more features as time goes on

- 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 at least, less so for low power

# Intel x86 Processors

- Machine Evolution

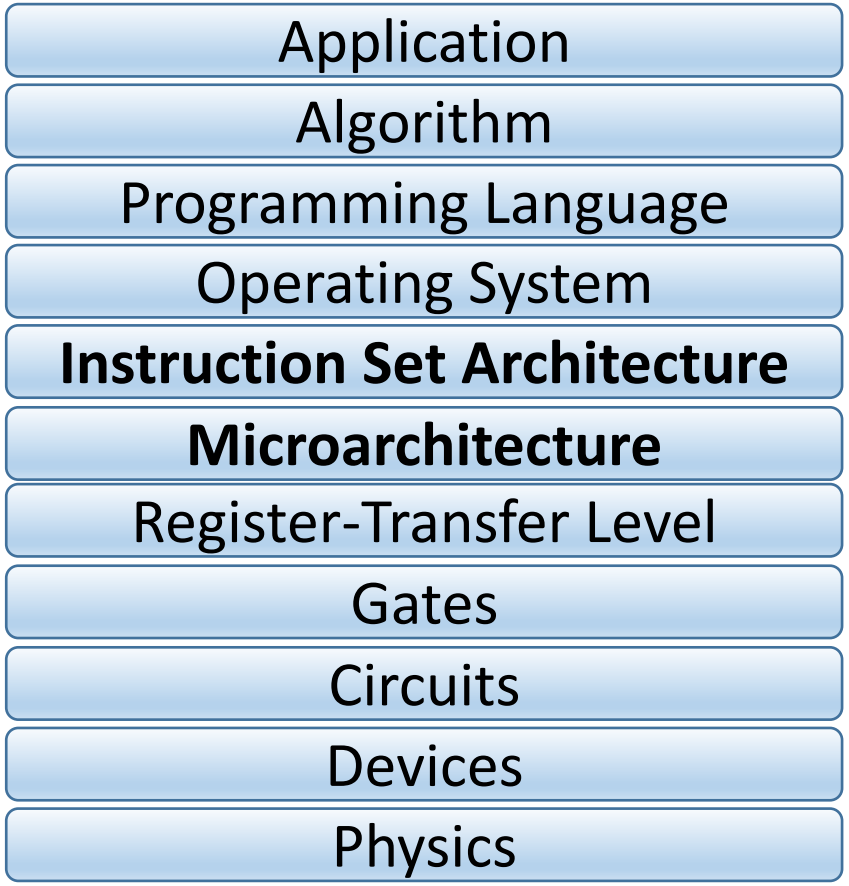
| Name | Date | Transistors | MHz |
|---|---|---|---|
| 8086 | 1979 | 29k | 5-10 |
| 386 | 1985 | 0.3M | 16-33 |
| Pentium | 1993 | 3.1M | 60-300 |
| Pentium 4 | 2000 | 45M | 1400-1500 |
| Core 2 Duo | 2006 | 291M | 1860-2670 |
| Core i7 | 2008 | 731M | 1700-3900 |
| Core i7 Skylake | 2015 | 1.75B | 2800-4000 |



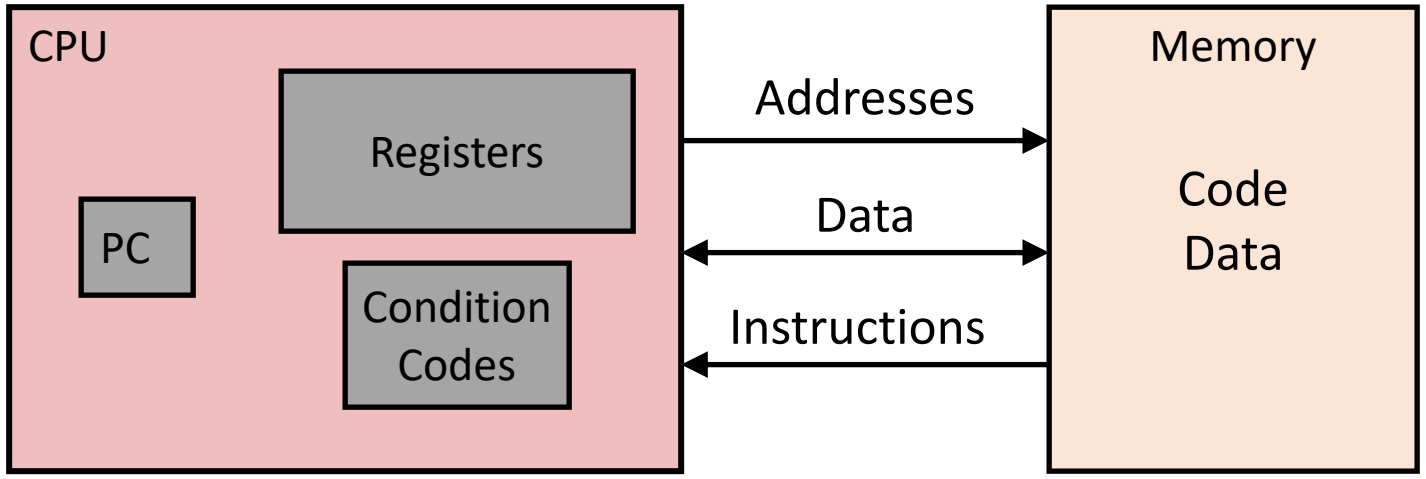

- Added Features
  - Instructions to support multimedia operations
  - Instructions to enable more efficient conditional operations
  - Transition from 32 bits to 64 bits
  - More cores
  - Built-in Graphics Processor

# Definitions

- Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code
  - Examples:  instruction set specification, registers
- Microarchitecture: Implementation of the architecture
  - Can have many microarchitectures implement the same ISA e.g., different cache sizes and core frequencies
- Code Forms:
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code
- Example ISAs:
  - Intel/AMD: IA32, x86-64
  - ARM: ARMv6, ARMv7E, ARMv8
  - RISC-V: RV32I, RV64I, RV64G



# Assembly/Machine Code View



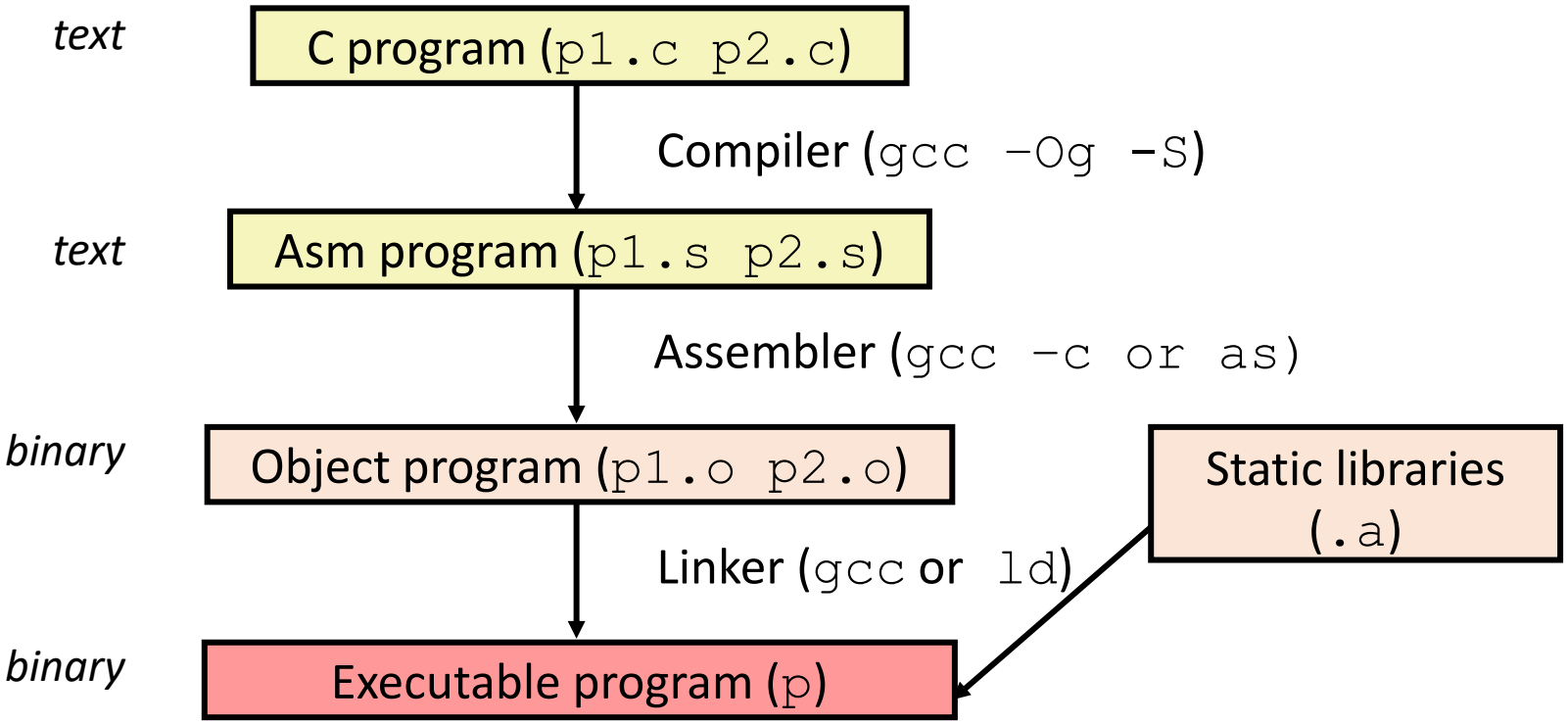

## Programmer-Visible State

- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching
- **PC: Program counter**
  - Address of next instruction
  - Called “RIP” (Instruction Pointer Register) in X86-64

- **Memory**
  - Byte addressable array
  - Code and user data

# Turning C into Object Code

- Code in files **`p1.c p2.c`**
- Compile with command: **`gcc –Og p1.c p2.c -o p`**
  - Use basic optimizations (**`–Og`**) [New to recent versions of GCC]
  - Put resulting binary in file **`p`**



# Compiling Into Assembly

C Code (mult_and_add.c)

```
long mult_and_add(long x, long y, long z) {
    long product = x * y;
    return z + product
}
```

Generated x86-64 Assembly

```
mult_and_add:
    imulq %rsi, %rdi
    leaq  (%rdi, %rdx), %rax
    retq
```

Obtain (on a lab machine) with command

```
gcc -Og -S mult_and_add.c
```

Produces the file `mult_and_add.s`

*Warning*: You will get very different results on other machines (e.g., MacOS) due to different versions of gcc and different compiler settings

# Assembly Characteristics: Data Types

- Integer data of 1 (char), 2 (short),  4 (int), or 8 (long) bytes
  - Data values (signed and unsigned)
  - Addresses (pointers)
- Floating point data of 4 (float) or 8 (double) bytes
  - Stored in a different set of registers
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

# Assembly Characteristics: Operations

- Perform arithmetic function on registers or memory data
  - Math and logic operations
- 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 `mult_and_add`

```
mult_and_add:
    imulq %rsi, %rdi
    leaq  (%rdi, %rdx), %rax
    retq
```

**0x48**
**0x0f**
**0xaf**
**0xfe**
0x48
0x8d
0x04
0x17
**0xc3**

- Assembler: `gcc -c mult_and_add.s`
  - Translates .s into .o
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
  - The number of bytes per instruction varies
- Linker
  - Resolves references between files
  - Combines with code from run-time libraries
    - E.g., code for `malloc()`, `printf()`
  - Some libraries are dynamically linked
    - Linking occurs when program begins execution

# Disassembling Object Code

- Disassembler

  **`objdump -d mult_and_add`**

  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either executable binary program or `.o` file

- Disassembled

```
00000000004004e7 <mult_and_add>:
  4004e7: 48 0f af fe           imul   %rsi,%rdi
  4004eb: 48 8d 04 17           lea    (%rdi,%rdx),%rax
  4004ef: c3                    retq
```

# x86-64 Integer Registers

| | |
|---|---|
| %rax | %r8 |
| %rbx | %r9 |
| %rcx | %r10 |
| %rdx | %r11 |
| %rsi | %r12 |
| %rdi | %r13 |
| %rsp | %r14 |
| %rbp | %r15 |

# x86-64 Integer Registers

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

- Can reference low-order 4 bytes

# Integer Registers

- The lower portion of each 64-bit register can referred to by alternate register names
- All 64-bit registers start with r
- All named 32-bit registers start with e

| 64-bit register | Lower 32 bits | Lower 16 bits | Lower 8 bits |
|---|---|---|---|
| rax | eax | ax | al |
| rbx | ebx | bx | bl |
| rcx | ecx | cx | cl |
| rdx | edx | dx | dl |
| rsi | esi | si | sil |
| rdi | edi | di | dil |
| rbp | ebp | bp | bpl |
| rsp | esp | sp | spl |
| r8 | r8d | r8w | r8b |
| r9 | r9d | r9w | r9b |
| r10 | r10d | r10w | r10b |
| r11 | r11d | r11w | r11b |
| r12 | r12d | r12w | r12b |
| r13 | r13d | r13w | r13b |
| r14 | r14d | r14w | r14b |
| r15 | r15d | r15w | r15b |

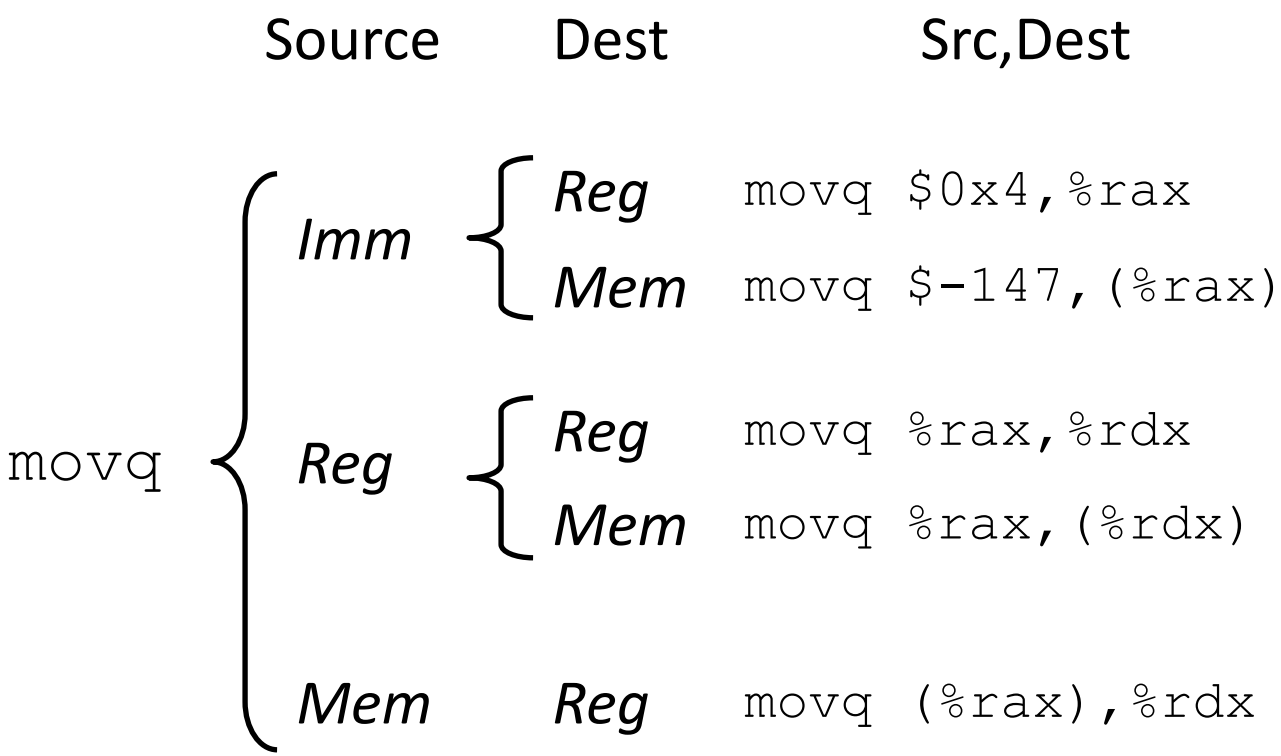
# Assembly instructions

- Instruction Format: **`ins`** ***`Source, Dest`***
  - **`ins`**: opcode (instruction)
  - **`source`**, **`dest`**: operands
    - Most opcodes have two operands, but some only have one
- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: **`$0x400`**, **`$-533`**
    - Like C constants, but prefixed with **`$`**
    - Encoded with either 1, 2, or 4 bytes
  - ***Register:*** One of 16 the integer register names prefixed with a `%`
    - Example: **`%rax`**, **`%r13`**
    - Some registers have special uses for particular instructions
  - ***Memory:*** Consecutive bytes of memory at a given address
    - Simplest example: **`(%rax)`**
    - Various other “addressing modes”
    - **Note**: an address can also be specified with as a constant without the $ prefix

# Our first instruction: move (mov)

- `movq Source, Dest`
  - Moves (copies) the source operand to the destination operand
  - Has many purposes
    - Load an immediate value (number) into a register
    - Copy a value from one register into another register
    - Read a value from a memory address
    - Write a value from a memory address
- In other hardware architectures, these operations are done with several different instructions

# `movq` Operand Combinations




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

# Instruction suffixes

- Most assembly instructions take a suffix:
  - b (byte: 1 byte)
  - w (word: 2 bytes)
  - l (long word: 4 bytes)
  - q (quad word: 8 bytes)
- Often used with the low-order registers (e.g., `%eax`, `%ax`, `%al`)
  - `movb $-17, %al`
  - `movw %bp, %sp`
  - `movl $0x4050, %eax`
- In general, only the specific register bytes or memory locations are modified
  - Exception: "`l`" instructions that have a register as a destination will set the upper order bits to 0

# Normal Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Pointer dereferencing in C

```
movq (%rcx),%rax
```

# Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Pointer dereferencing in C

```
movq (%rcx),%rax
```

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

```
movq 8(%rbp),%rdx
```

# Indexed Memory Addressing Modes

- Indexed ($R_b$, $R_i$) Mem[Reg[$R_b$] + Reg[$R_i$]]
  - Register $R_b$ often specifies base memory address
  - Register $R_i$ often acts as an index
  - Often used in accessing arrays

```
movq (%rcx, %rdx),%rax
```

- Scaled Indexed ($R_b$, $R_i$, s) Mem[Reg[$R_b$] + Reg[$R_i$]*s]
  - s is called the scaling factor
  - Must be 1, 2, 4, 8 (why these numbers?)
  - `movq (%rcx, %rdx, 8),%rax`
  - $R_b$ is optional ( , $R_i$, s) is a valid operand

# Complete Memory Addressing Modes

- Most General Form

$$D(R_b,R_i,S) \qquad Mem[Reg[R_b]+S*Reg[R_i]+ D]$$

  - D: Constant “displacement” 1, 2, or 4 bytes
  - $R_b$: Base register: Any of 16 integer registers
  - $R_i$: Index register: Any, except for **`%rsp`**
  - S: Scale: 1, 2, 4, or 8

# Address Computation Examples

| | |
|---|---|
| `%rdx` | `0xf000` |
| `%rcx` | `0x0100` |

| Expression | Address Computation | Address |
|---|---|---|
| `0x8(%rdx)` | | |
| `(%rdx,%rcx)` | | |
| `(%rdx,%rcx,4)` | | |
| `0x80(,%rdx,2)` | | |

# Address Computation Examples

| | |
|---|---|
| `%rdx` | `0xf000` |
| `%rcx` | `0x0100` |

| Expression | Address Computation | Address |
|---|---|---|
| `0x8(%rdx)` | `0xf000 + 0x8` | `0xf008` |
| `(%rdx,%rcx)` | | |
| `(%rdx,%rcx,4)` | | |
| `0x80(,%rdx,2)` | | |

# Address Computation Examples

| | |
|---|---|
| `%rdx` | `0xf000` |
| `%rcx` | `0x0100` |

| Expression | Address Computation | Address |
|---|---|---|
| `0x8(%rdx)` | `0xf000 + 0x8` | `0xf008` |
| `(%rdx,%rcx)` | `0xf000 + 0x100` | `0xf100` |
| `(%rdx,%rcx,4)` | | |
| `0x80(,%rdx,2)` | | |

# Address Computation Examples

| | |
|---|---|
| %rdx | 0xf000 |
| %rcx | 0x0100 |

| Expression | Address Computation | Address |
|---|---|---|
| 0x8(%rdx) | 0xf000 + 0x8 | 0xf008 |
| (%rdx,%rcx) | 0xf000 + 0x100 | 0xf100 |
| (%rdx,%rcx,4) | 0xf000 + 4*0x100 | 0xf400 |
| 0x80(,%rdx,2) | | |

# Address Computation Examples

| | |
|---|---|
| %rdx | 0xf000 |
| %rcx | 0x0100 |

| Expression | Address Computation | Address |
|---|---|---|
| 0x8(%rdx) | 0xf000 + 0x8 | 0xf008 |
| (%rdx,%rcx) | 0xf000 + 0x100 | 0xf100 |
| (%rdx,%rcx,4) | 0xf000 + 4*0x100 | 0xf400 |
| 0x80(,%rdx,2) | 2*0xf000 + 0x80 | 0x1e080 |

# Address Computation Examples

| Operand | Address | Value at Address |
|---|---|---|
| `0x104` | | |
| `(%rax)` | | |
| `4(%rax)` | | |
| `9(%rax, %rdx)` | | |
| `260(%rcx, %rdx)` | | |
| `0xFC(, %rcx, 4)` | | |
| `(%rax, %rdx, 4)` | | |

| Register | Value |
|---|---|
| `%rax` | `0x100` |
| `%rcx` | `0x1` |
| `%rdx` | `0x3` |

| Memory Address | Value |
|---|---|
| `0x100` | `0xFF` |
| `0x104` | `0xAB` |
| `0x108` | `0x13` |
| `0x10C` | `0x11` |

# Address Computation Examples

| Operand | Address | Value at Address |
|---|---|---|
| `0x104` | `0x104` | `0xAB` |
| `(%rax)` | | |
| `4(%rax)` | | |
| `9(%rax, %rdx)` | | |
| `260(%rcx, %rdx)` | | |
| `0xFC(, %rcx, 4)` | | |
| `(%rax, %rdx, 4)` | | |

| Register | Value |
|---|---|
| `%rax` | `0x100` |
| `%rcx` | `0x1` |
| `%rdx` | `0x3` |

| Memory Address | Value |
|---|---|
| `0x100` | `0xFF` |
| `0x104` | `0xAB` |
| `0x108` | `0x13` |
| `0x10C` | `0x11` |

# Address Computation Examples

| Operand | Address | Value at Address |
|---|---|---|
| `0x104` | `0x104` | `0xAB` |
| `(%rax)` | `0x100` | `0xFF` |
| `4(%rax)` | | |
| `9(%rax, %rdx)` | | |
| `260(%rcx, %rdx)` | | |
| `0xFC(, %rcx, 4)` | | |
| `(%rax, %rdx, 4)` | | |

| Register | Value |
|---|---|
| `%rax` | `0x100` |
| `%rcx` | `0x1` |
| `%rdx` | `0x3` |

| Memory Address | Value |
|---|---|
| `0x100` | `0xFF` |
| `0x104` | `0xAB` |
| `0x108` | `0x13` |
| `0x10C` | `0x11` |

# Address Computation Examples

| Operand | Address | Value at Address |
|---|---|---|
| `0x104` | `0x104` | `0xAB` |
| `(%rax)` | `0x100` | `0xFF` |
| `4(%rax)` | `0x104` | `0xAB` |
| `9(%rax, %rdx)` | | |
| `260(%rcx, %rdx)` | | |
| `0xFC(, %rcx, 4)` | | |
| `(%rax, %rdx, 4)` | | |

| Register | Value |
|---|---|
| `%rax` | `0x100` |
| `%rcx` | `0x1` |
| `%rdx` | `0x3` |

| Memory Address | Value |
|---|---|
| `0x100` | `0xFF` |
| `0x104` | `0xAB` |
| `0x108` | `0x13` |
| `0x10C` | `0x11` |

# Address Computation Examples

| Operand | Address | Value at Address |
|---|---|---|
| `0x104` | `0x104` | `0xAB` |
| `(%rax)` | `0x100` | `0xFF` |
| `4(%rax)` | `0x104` | `0xAB` |
| `9(%rax, %rdx)` | `0x100 + 3 + 9 = 0x10C` | `0x11` |
| `260(%rcx, %rdx)` | | |
| `0xFC(, %rcx, 4)` | | |
| `(%rax, %rdx, 4)` | | |

| Register | Value |
|---|---|
| `%rax` | `0x100` |
| `%rcx` | `0x1` |
| `%rdx` | `0x3` |

| Memory Address | Value |
|---|---|
| `0x100` | `0xFF` |
| `0x104` | `0xAB` |
| `0x108` | `0x13` |
| `0x10C` | `0x11` |

# Address Computation Examples

| Operand | Address | Value at Address |
|---|---|---|
| 0x104 | 0x104 | 0xAB |
| (%rax) | 0x100 | 0xFF |
| 4(%rax) | 0x104 | 0xAB |
| 9(%rax, %rdx) | 0x100 + 3 + 9 = 0x10C | 0x11 |
| 260(%rcx, %rdx) | 0x104 + 1 + 3 = 0x108 | 0x13 |
| 0xFC(, %rcx, 4) | | |
| (%rax, %rdx, 4) | | |

| Register | Value |
|---|---|
| %rax | 0x100 |
| %rcx | 0x1 |
| %rdx | 0x3 |

| Memory Address | Value |
|---|---|
| 0x100 | 0xFF |
| 0x104 | 0xAB |
| 0x108 | 0x13 |
| 0x10C | 0x11 |

# Address Computation Examples

| Operand | Address | Value at Address |
|---|---|---|
| `0x104` | `0x104` | `0xAB` |
| `(%rax)` | `0x100` | `0xFF` |
| `4(%rax)` | `0x104` | `0xAB` |
| `9(%rax, %rdx)` | `0x100 + 3 + 9 = 0x10C` | `0x11` |
| `260(%rcx, %rdx)` | `0x104 + 1 + 3 = 0x108` | `0x13` |
| `0xFC(, %rcx, 4)` | `0xFC + 1*4 = 0x100` | `0xFF` |
| `(%rax, %rdx, 4)` | | |

| Register | Value |
|---|---|
| `%rax` | `0x100` |
| `%rcx` | `0x1` |
| `%rdx` | `0x3` |

| Memory Address | Value |
|---|---|
| `0x100` | `0xFF` |
| `0x104` | `0xAB` |
| `0x108` | `0x13` |
| `0x10C` | `0x11` |

# Address Computation Examples

| Operand | Address | Value at Address |
|---|---|---|
| `0x104` | `0x104` | `0xAB` |
| `(%rax)` | `0x100` | `0xFF` |
| `4(%rax)` | `0x104` | `0xAB` |
| `9(%rax, %rdx)` | `0x100 + 3 + 9 = 0x10C` | `0x11` |
| `260(%rcx, %rdx)` | `0x104 + 1 + 3 = 0x108` | `0x13` |
| `0xFC(, %rcx, 4)` | `0xFC + 1*4 = 0x100` | `0xFF` |
| `(%rax, %rdx, 4)` | `0x100 + 3*4 = 0x10c` | `0x11` |

| Register | Value |
|---|---|
| `%rax` | `0x100` |
| `%rcx` | `0x1` |
| `%rdx` | `0x3` |

| Memory Address | Value |
|---|---|
| `0x100` | `0xFF` |
| `0x104` | `0xAB` |
| `0x108` | `0x13` |
| `0x10C` | `0x11` |

# Machine Programming Basics: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms