Computer Systems Organization

Topic 3

Based on chapter 3 from Computer Systems by Randal E. Bryant and David R. O'Hallaron

Historical Perspective: Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- x86 is a family of complex instruction set computer (CISC) instruction set architectures
  - Initially developed by Intel based on the Intel 8086 microprocessor and its 8088 variant
  - 8086 was introduced in 1978 as a fully 16-bit extension of Intel's 8-bit 8080 microprocessor
  - The term "x86" came into being because the names of several successors to Intel's 8086 processor end in "86", including the 80816, 80286, 80386 and 80486 processors.

Intel x86 Processors

- As of 2022, most desktop computers, laptops and game consoles (exception of Nintendo Switch) sold are based on the x86 architecture family, while mobile categories such as smartphones or tablets are dominated by ARM processor developed by Advance RISC Machines (ARM).
- At the high end, x86 continues to dominate compute intensive workstation and cloud computing segments, while the fastest supercomputer in 2020 was ARM-based, with the top 4 no longer x86-based in that year.
- Wiki page provides list of X86 CPUs

Intel x86 Processors (wiki page)

- 16 bit: 8086, 8088, 80186, 80286
- 32 bit Intel: i386 (80386), i486 (80486), Pentium, Pentium Pro, Pentium 2 (or II), Pentium 3 (or III), Older versions of the Pentium 4, Pentium M, Core, Older Xeon, Mobile versions of Intel Atom, Older Celeron
- 64 bit Intel: Newer Prescott Pentium 4, Pentium D, Core 2, Core i3, i5, i7, and i9, Newer Atom, Pentium dual core, Newer Celeron, Newer Xeon
- 32 bit AMD: AMD386, AMD486, AMD586, Am5x86-P75, K5, K6/K6-II/K6-III, Athlon, Athlon XP, Duron, Sempron, Geode

Intel x86 Processors (wiki page)

- 64 bit AMD: Opteron, Athlon 64, Phenom, Phenom 2, FX, Sempron, APU A4/A6/A8/A10/A12, APU Athlon, APU Sempron, Ryzen, Epyc
- Other processors: Cyrix 386/486S/DLC, 5x86, 6x86, MII, MIII (32 bit), IDT Winchip (32 bit), Rise (32 bit), NXGen (32 bit), Via C3 and C7 (32 bit), Via Nano (64 bit)

Instruction set

- An instruction set is a group of commands for a CPU in machine language. The term can refer to all possible instructions for a CPU or a subset of instructions to enhance its performance in certain situations. Some instructions are simple read, write and move commands that direct data to different hardware.
- A complex instruction set computer (CISC) is a computer architecture in which single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions.
  - Term retroactively coined in contrast to reduced instruction set computer (RISC) and has become umbrella term for everything that is not RISC
  - Most RISC designs use uniform instruction length for almost all instructions, and employ strictly separate load and store instructions.

Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
8086	**1978**	**29K**	**5-10**
• First 16-bit Intel processor. Basis for IBM PC & DOS			
• 1MB address space			
386	**1985**	**275K**	**16-33**
• First 32 bit Intel processor , referred to as IA32			
• Added “flat addressing”, capable of running Unix			
Pentium 4E	**2004**	**125M**	**2800-3800**
• First 64-bit Intel x86 processor, referred to as x86-64			
Core 2	**2006**	**291M**	**1060-3500**
• First multi-core Intel processor			
Core i7	**2008**	**731M**	**1700-3900**
• Four cores			

Address space vs. Memory space

- Address used by programmer is a virtual address - set of such addresses generated by the programs as they reference instructions and data is the address space. Generally, address space is larger than the memory space.
- An address in main memory is called a location or physical address. Set of such locations is called the memory space. Generally, the memory space is smaller than the address space.
- For example if,
  - Address space = 24 bits
  - Memory space = 16 bits
  - # of address space possible = 2^24 = 16M
  - # of memory spaces possible = 2^16 = 64K

CPU Clock Speed

- Performance of CPU (“brain” of your PC) has a major impact on the speed at which programs load and how smoothly they run.
- Few different ways to measure processor performance - clock speed (also “clock rate” or “frequency”) is one of the most significant. A higher clock speed in general means a faster CPU. However, many other factors come into play.
- CPU processes many instructions (low-level calculations like arithmetic) from different programs every second. The clock speed measures the number of cycles CPU executes per second, measured in GHz (gigahertz).
- A “cycle” is technically a pulse synchronized by an internal oscillator, but for our purposes, they’re a basic unit that helps understand a CPU’s speed.

CPU Clock Speed

- A CPU with a clock speed of 3.2 GHz executes 3.2 billion cycles per second (older CPUs had speeds measured in megahertz, or millions of cycles per second). Sometimes, multiple instructions are completed in a single clock cycle; in other cases, one instruction might be handled over multiple clock cycles.
- Since different CPU designs handle instructions differently, it’s best to compare clock speeds within the same CPU brand and generation e.g., a CPU with a higher clock speed from 5 years ago might be outperformed by a new CPU with a lower clock speed, as newer architecture may deal with instructions more efficiently.
- Within the same generation of CPUs, a processor with a higher clock speed will generally outperform a processor with a lower clock speed across many applications. In general, CPU clock speed is a good indicator of your processor’s performance.

Intel x86 Processors, cont.

- Machine Evolution (in terms of # of transistors)

386	1985	0.3M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7	2008	731M

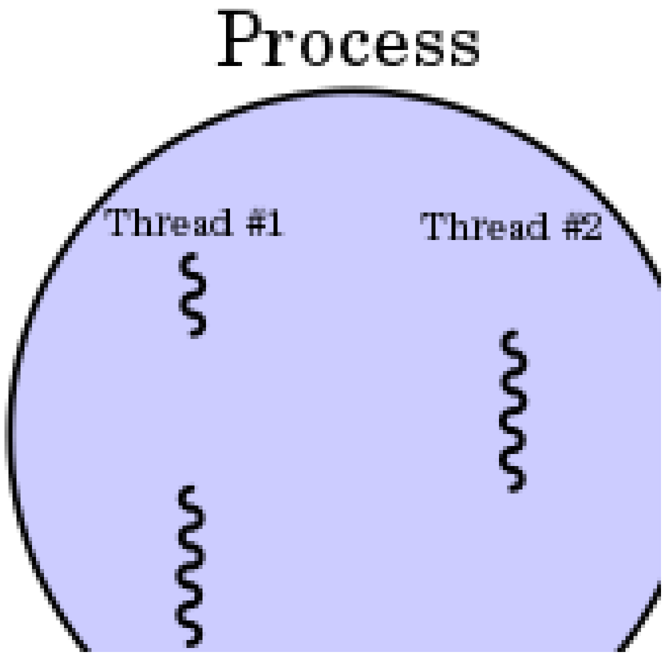
- Added Features
  - Instructions to support multimedia operations
  - Instructions to enable more efficient conditional operations
  - Transition from 32 bits to 64 bits
  - More cores

Multi core processor

- A multi-core processor is a computer processor with two or more separate processing units, called cores, each of which reads and executes program instructions. The instructions are ordinary CPU instructions (such as add, move data, and branch) but the single processor can run instructions on separate cores at the same time.
- Multithreading is the ability of a CPU (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system. This approach differs from multiprocessing since the threads share the resources of a single or multiple cores, which include the computing units, the CPU caches, and the translation lookaside buffer (TLB).

Multithreading

- Multiprocessing systems include multiple complete processing units while multithreading aims to increase utilization of a single core by using thread-level parallelism, as well as instruction-level parallelism. Since the two techniques are complementary, they are combined in nearly all modern systems architectures.



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
- In recent Years Intel got its act together and leads the world in semiconductor technology

Intel’s 64-Bit History

- 2001: Intel Attempts Radical Shift from IA32 to IA64 (IA is Intel Architecture)
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- 2003: AMD Steps 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 except low-end x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode

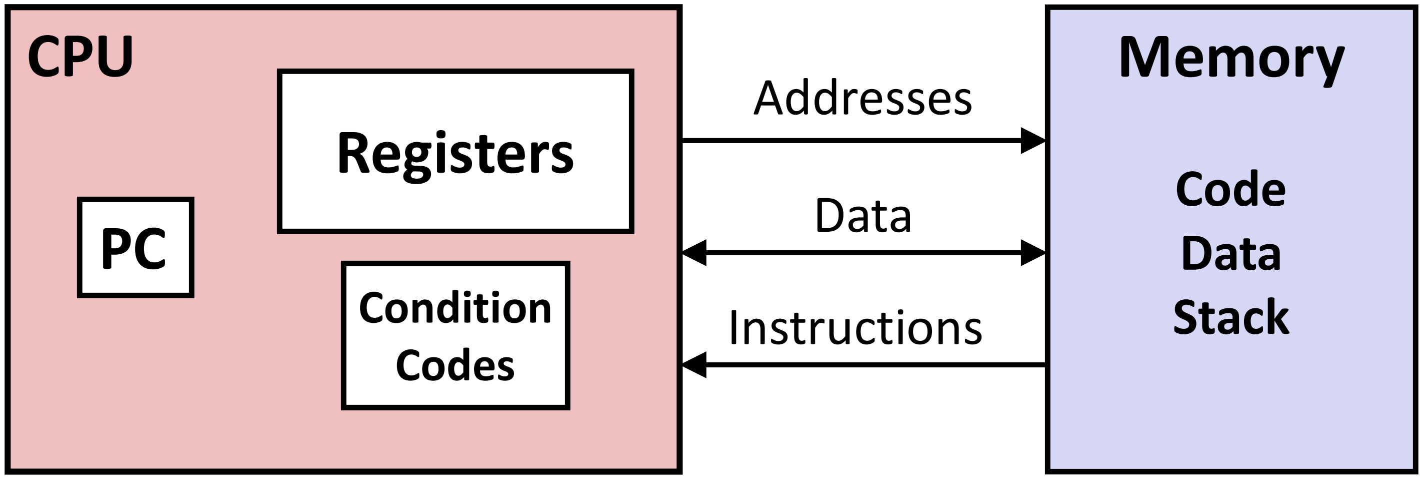
Definitions

- **Architecture (also ISA: instruction set architecture):** ISA is part of the abstract model of a computer that defines how the CPU is controlled by the software. It is a structure of commands and operations used by software to communicate with hardware. Acts as an interface between the hardware and software, specifying both what the processor is capable of doing as well as how it gets done e.g., x86 instruction set
    - ISA provides the only way through which a user is able to interact with the hardware - can be viewed as a programmer's manual because it's the portion of the machine that's visible to the assembly language programmer, the compiler writer, and the application programmer.
    - ISA defines the supported data types, the registers, how the hardware manages main memory, key features (such as virtual memory) etc. ISA can be extended by adding instructions or other capabilities, or by adding support for larger addresses and data values.

Definitions

- **Microarchitecture** (also called computer organization and sometimes abbreviated as µarch or uarch) is a hardware implementation of an ISA (Instruction Set Architecture) i.e., it is the hardware circuitry that implements one particular ISA
  - A given ISA maybe implemented with different microarchitectures
  - Multiple CPU models may be designed for a particular microarchitecture. For this reason, a microarchitecture is sometimes referred to as a "family" or "generation" of CPU. For example, Intel Kaby Lake (7th generation) and Coffee Lake (8th generation) are separate microarchitectures, each with a "family" of compatible CPUs.
- **Computer architecture** is the combination of microarchitecture and instruction set architecture.
  - Code Forms include Machine Code (byte-level programs that a processor executes) and Assembly Code (text representation of machine code)

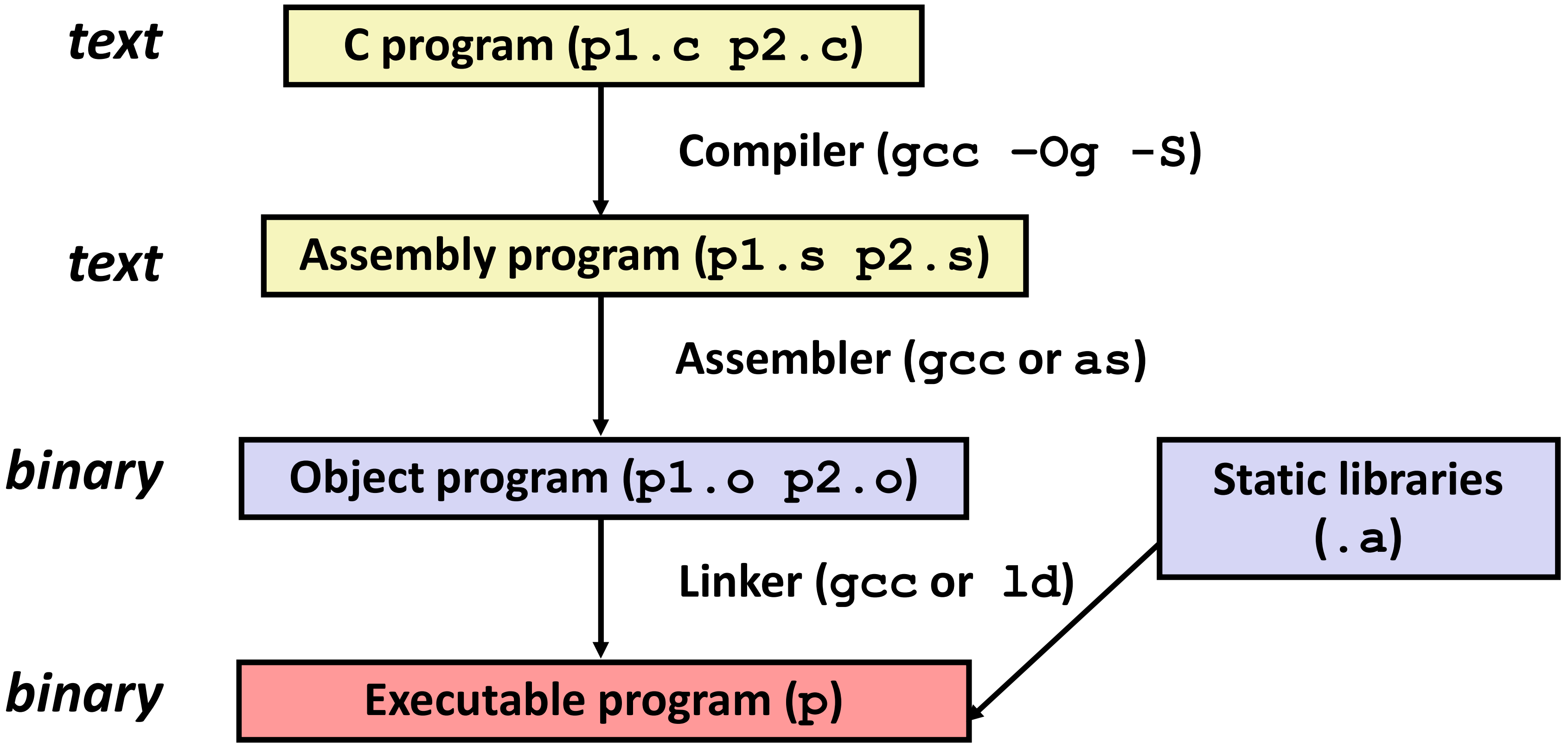
Assembly/Machine Code View of Processor State



- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64) [Register for Instruction Pointer]
- **Register file**
  - Holds addresses (pointers) or integer data
  - Heavily used program data
- **Condition code registers**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching
- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

Turning C into Object Code

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



Compiling Into Assembly

C Code (mstore.c)

```
long mult2(long, long);

void multstore(long x, long y,
               long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
multstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    mult2
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

Obtained with command

```
gcc –Og –S mstore.c
```

Produces assembly file `mstore.s`

***Warning*: Can get very different results on different machines due to different versions of gcc and different compiler settings.**

Assembly Description

- Assembly code file mstore.s contains various declarations, including lines from previous slide
- Each line of the code corresponds to a single machine instruction
- For example, pushq instruction indicates that the contents of the register %rbx should be pushed onto the program stack
- All information about local variable names or data types has been stripped away

Assembly Characteristics: Data Types

- “Integer” data of 1 (char), 2 (short), 4 (int), or 8 (long, char *) bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4 (float), 8 (double), or 10 (long double) bytes
- 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 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

Embedded within the object code `mstore.o`

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0x00
    0x00
    0x00
    0x00
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- **Total of 14 bytes**
- **Each instruction 1 to 5 bytes each**
- **Starts at address `0x0400595`**

- **Assembler**
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
  - `gcc -Og -c mstore.c` (both compile and assemble)
- **Linker**
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for **`malloc`**, **`printf`**
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

Machine Instruction Example

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:   48 89 03
```

- **C Code**
  - Store value `t` where designated by `dest`
- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - `t`: Register `%rax`
    - `dest`: Register `%rbx`
    - `*dest`: Memory **M[`%rbx`]**
- **Object Code**
  - 3-byte instruction
  - Stored at address `0x40059e`

Disassembling Object Code

Disassembled

```
0000000000400595 <multstore>:
  400595:  53                 push   %rbx
  400596:  48 89 d3           mov    %rdx,%rbx
  400599:  e8 00 00 00 00     callq  400590 <mult2>
  40059e:  48 89 03           mov    %rax,(%rbx)
  4005a1:  5b                 pop    %rbx
  4005a2:  c3                 retq
```

- **Disassembler**
  - **`objdump -d mstore.o`**
  - Useful tool for examining object code
  - Produces approximate rendition of assembly code based purely on the byte sequences in the machine code file
  - Can be run on either `a.out` (complete executable) or `.o` file

Object Code

```
0000000000000000   <multstore>:
 Offset   Byte value          Equivalent assembly
     0:    53                  pushq   %rbx
     1:    48 89 d3            movq    %rdx, %rbx
     4:    e8 00 00 00 00      call    mult2
     9:    48 89 03            movq    %rax, (%rbx)
     c:    5b                  popq    %rbx
     d:    c3                  ret
```

- X86-64 instructions can range in length from 1 to 15 bytes

- Encoding is done so commonly used and those with fewer operands require a smaller number of bytes

- Disassembler determines assembly code based purely on byte sequences (does not require access to source/assembly)

Assembly Basics: x86-64 Integer Registers

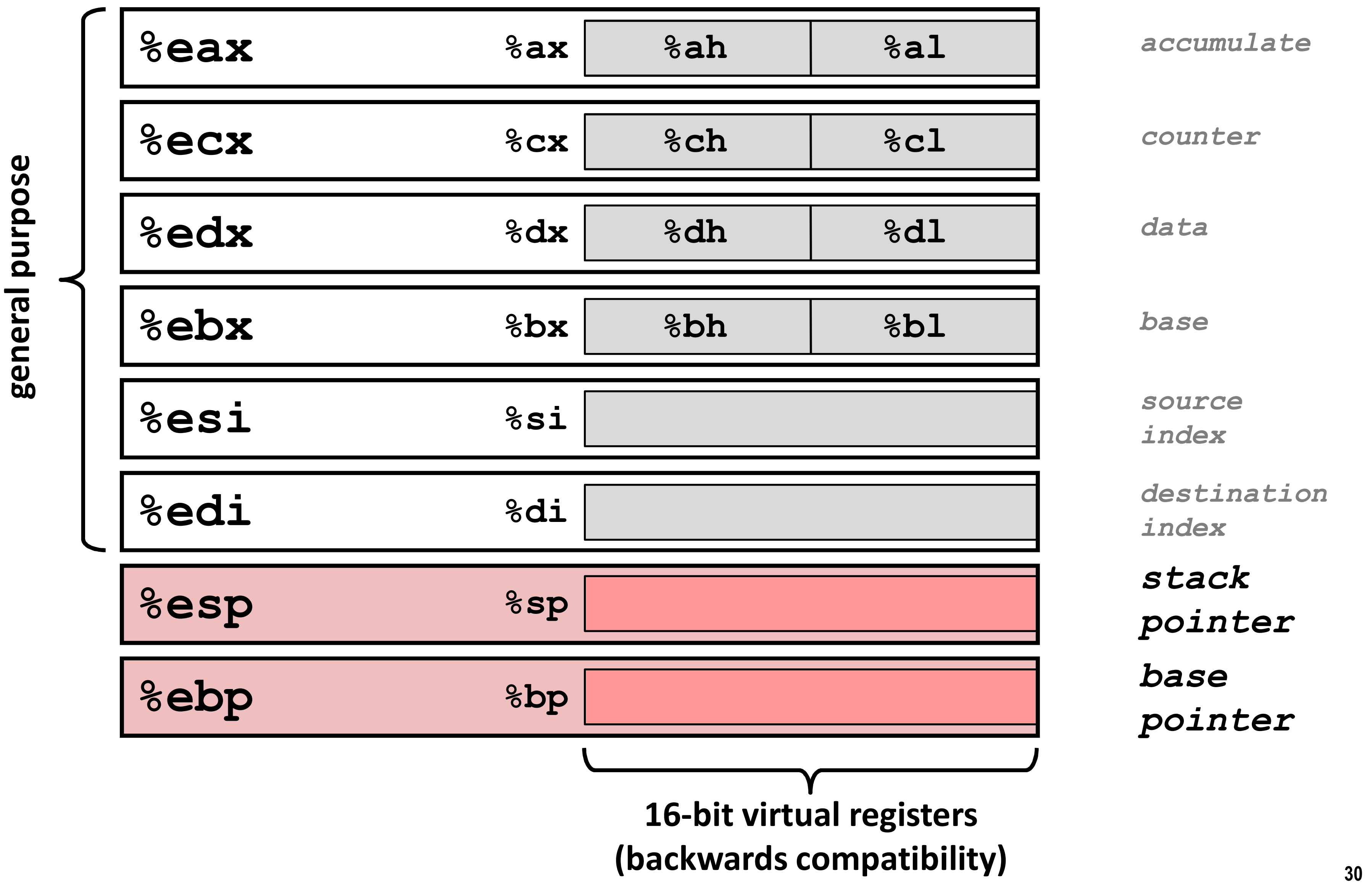
`%rax`	`%eax`	`%r8`	`%r8d`
`%rbx`	`%ebx`	`%r9`	`%r9d`
`%rcx`	`%ecx`	`%r10`	`%r10d`
`%rdx`	`%edx`	`%r11`	`%r11d`
`%rsi`	`%esi`	`%r12`	`%r12d`
`%rdi`	`%edi`	`%r13`	`%r13d`
`%rsp`	`%esp`	`%r14`	`%r14d`
`%rbp`	`%ebp`	`%r15`	`%r15d`

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Please refer page 216, table 3.2

What do they stand for ?

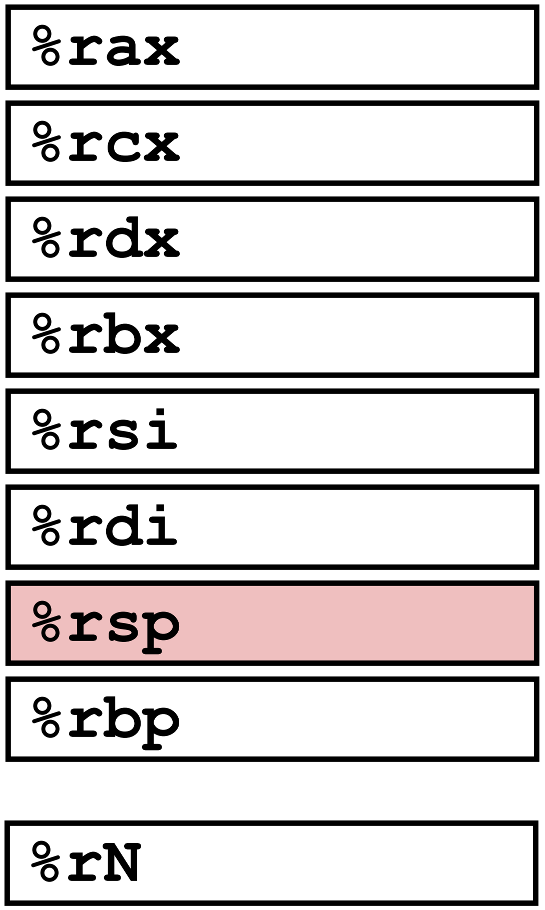
- 16 general purpose registers storing 64 bit values
- Store integer data as well as pointers
- Original 8086 had eight 16 bit registers %ax to %bp
- For IA32, registers expanded to 32 bit labeled %eax to %ebp
- For x86-64, the original eight registers were expanded to 64 bits, labeled %rax to %rbp. In addition, 8 new registers were added %r8 through %r15
- Stack pointer %rsp used to indicate end position in run time stack
- Other registers more flexible in usage

Some History: IA32 Registers



Moving Data

- Moving Data
  - **`movq`** ***Source*, *Dest***:
- Operand Types
  - *Immediate:* Constant integer data
    - Example: **`$0x400`**, **`$-533`**
    - Like C constant, but prefixed with **`'$'`**
    - Encoded with 1, 2, or 4 bytes
  - *Register:* One of the 16 integer registers
    - Example: **`%rax, %r13`**
    - But **`%rsp`** reserved for special use
  - *Memory:* 8 consecutive bytes of memory at address given by register
    - Simplest example: **`(%rax)`**
    - Various other "address modes"



- Source values can be constants or read from registers or memory. Results can be stored in registers or memory.

`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

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

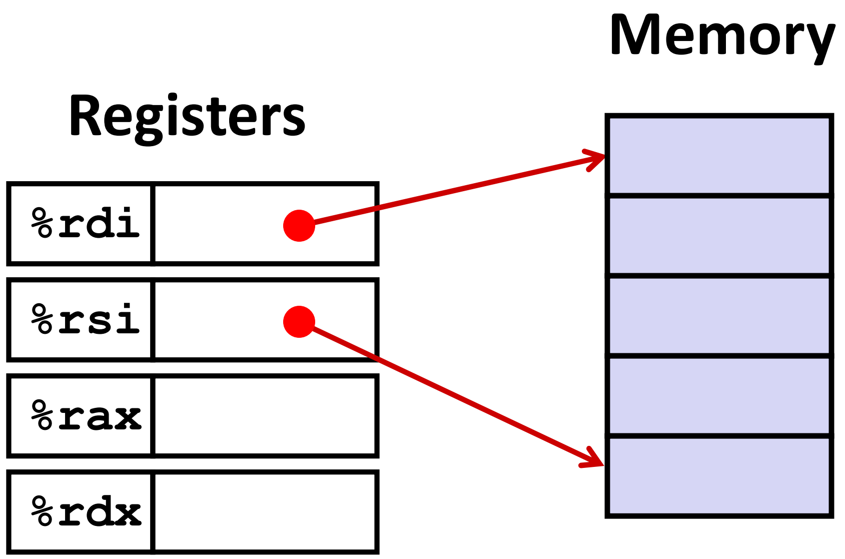
Example of Simple Addressing Modes

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

```
swap:
   movq    (%rdi), %rax
   movq    (%rsi), %rdx
   movq    %rdx, (%rdi)
   movq    %rax, (%rsi)
   ret
```

Understanding Swap()

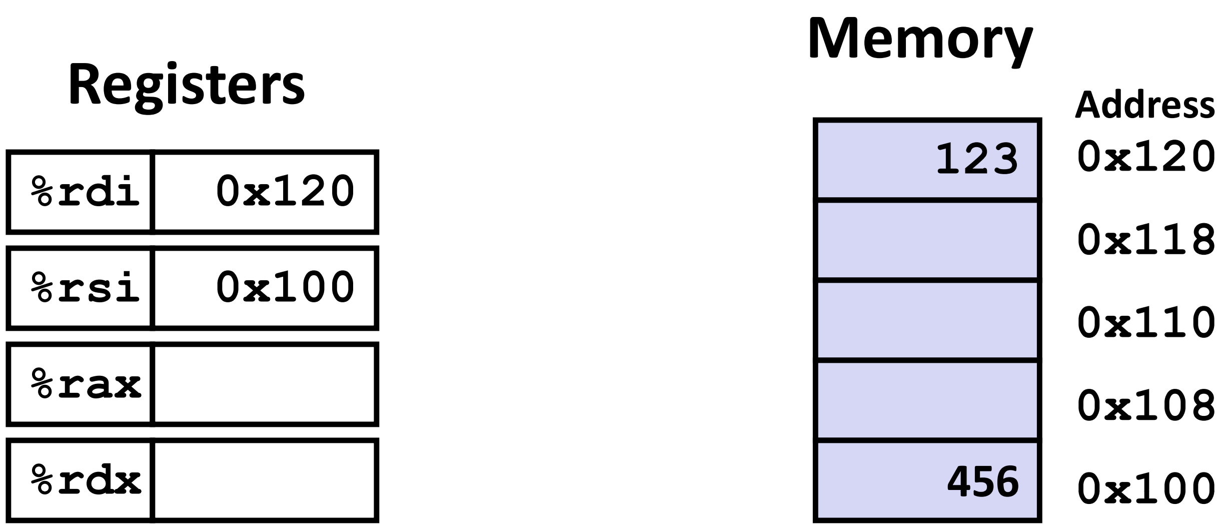
```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

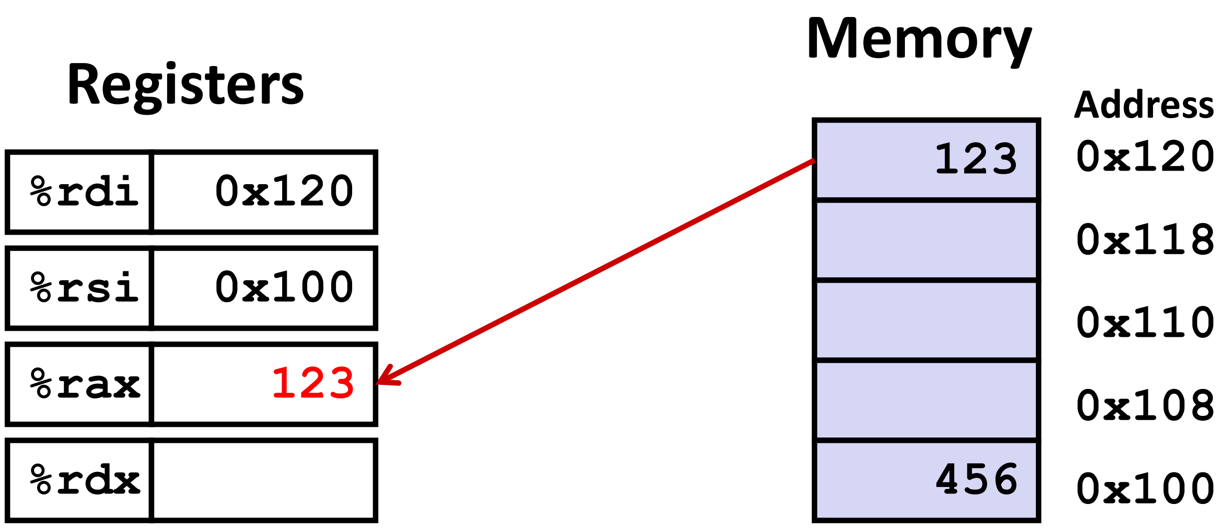
```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



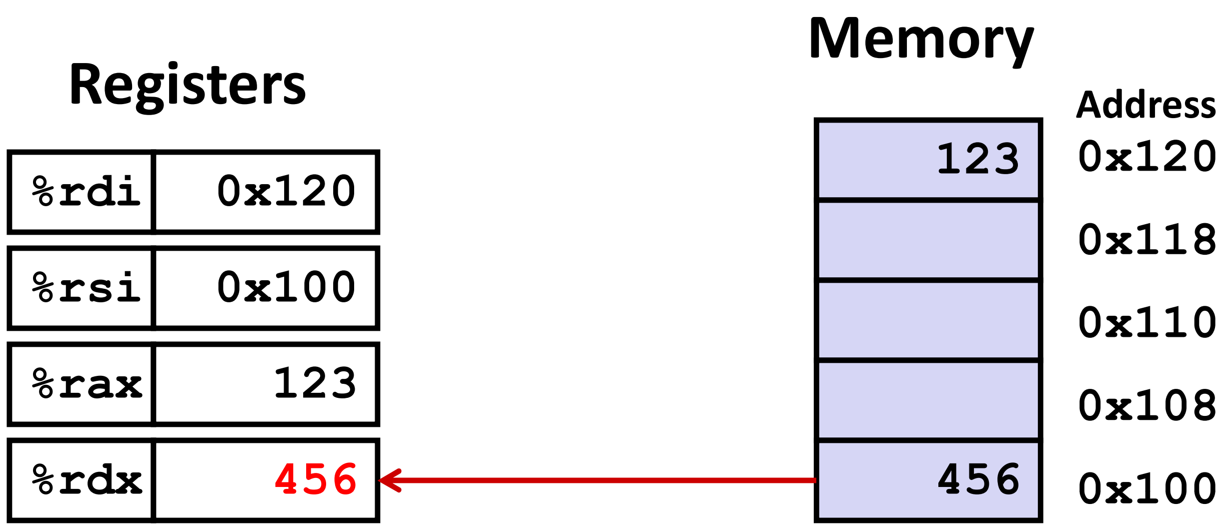
```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Understanding Swap()



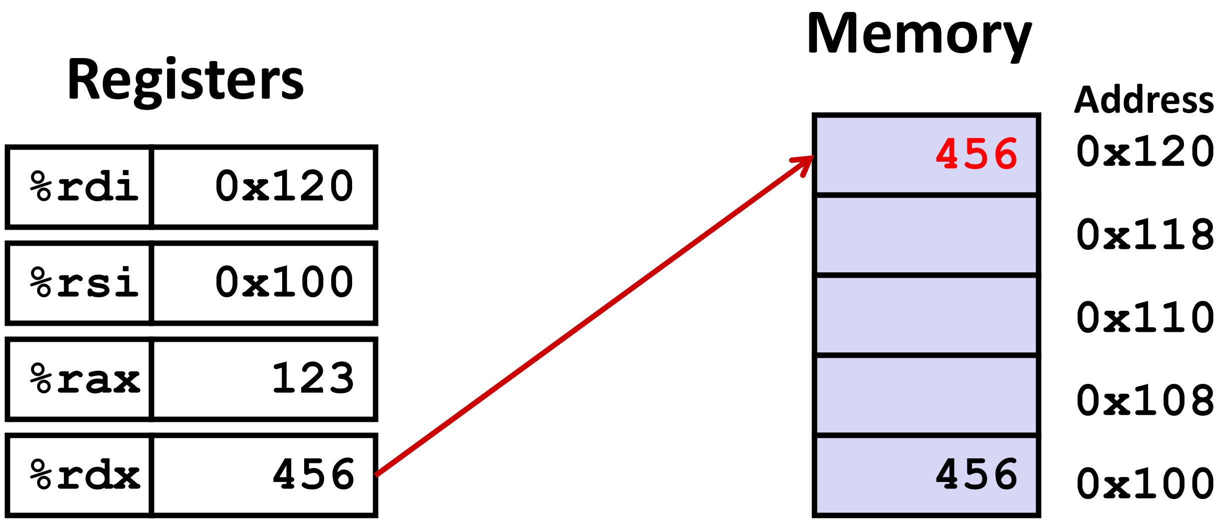
```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



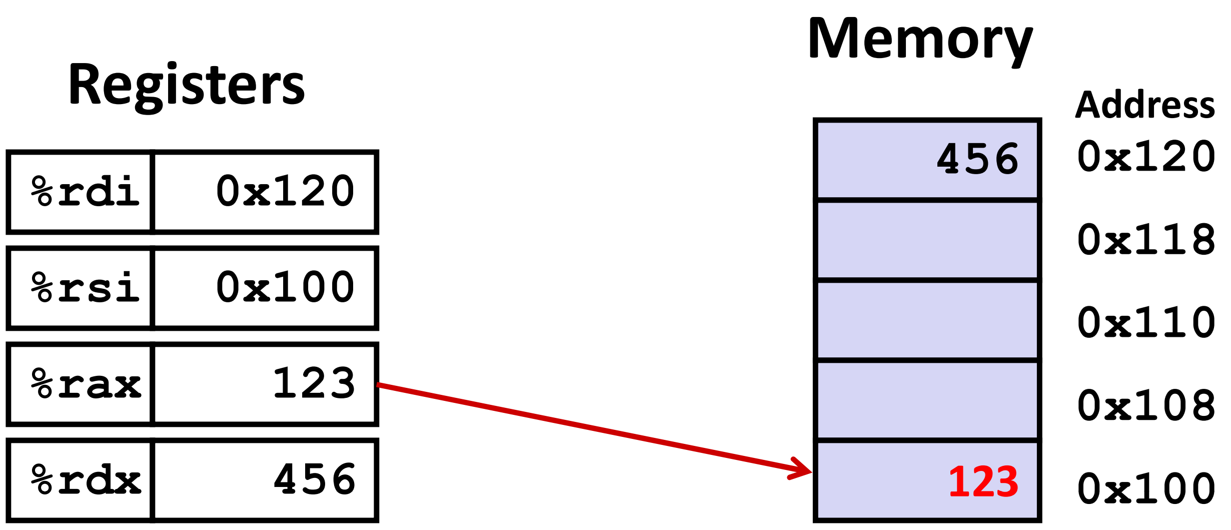
```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

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
```

Complete Memory Addressing Modes

- Most General Form
- **D(Rb,Ri,S)** **Mem[Reg[Rb]+S*Reg[Ri]+ D]**
  - D: Constant “displacement” of 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for **`%rsp`**
  - S: Scaling factor S must be 1, 2, 4, or 8

- Different Forms
- **(Rb,Ri)** **Mem[Reg[Rb]+Reg[Ri]]**
- **D(Rb,Ri)** **Mem[Reg[Rb]+Reg[Ri]+D]**
- **(Rb,Ri,S)** **Mem[Reg[Rb]+S*Reg[Ri]]**

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`

Zero extending data movement

- Two classes of data movement instructions when copying a smaller source value to a larger destination
  - Move with zero extension
  - Move with sign extension
- Move with zero extension MOVZ
  - movzbw → Byte to Word
  - movzbl → Byte to Double word
  - movzwl → Word to Double word
  - movzbq → Byte to Quad word
  - movzwq → Word to Quad word

Sign extending data movement

- Move with zero extension MOVS
  - movsbw → Byte to Word
  - movsbl → Byte to Double word
  - movswl → Word to Double word
  - movsbq → Byte to Quad word
  - movswq → Word to Quad word
  - movslq → Double word to Quad word

Arithmetic and Logical Operations

- Operations divided into 4 groups
  - load effective address (leaq), unary, binary and shifts
  - Operations given as instruction classes since they can have variants with different operand sizes (only leaq has no other size variant)
  - E.g., ADD consists of four additional instructions: addb, addw, addl, addq [bytes, words, double, quad]
- Leaq variant of movq – reads from memory to register
  - First operand appears to be a memory reference but instead of reading from the designated location, the instruction copies the effective address to the destination
  - Useful to generate pointer for later memory references and to compactly describe common arithmetic operations

Address Computation Instruction

- leaq Src, Dst
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression

- Uses
  - Computing addresses without a memory reference
    - E.g., translation of **`p = &x[i]`***;*
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8

- Example: **leaq 7(%rdx,%rdx,4), %rax**
  - Will set register %rax to 5x+7, if %rdx contains value x

Address Computation Instruction

- **Example**

```
long m12(long x)
{
  return x*12;
}
```

Upon compiling:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Some Arithmetic Operations

- Two Operand Instructions:

Format		*Computation*	
`addq`	*Src,Dest*	Dest = Dest + Src	
`subq`	*Src,Dest*	Dest = Dest – Src	
`imulq`	*Src,Dest*	Dest = Dest * Src	
`salq`	*Src,Dest*	Dest = Dest << Src	***Also called shlq***
`sarq`	*Src,Dest*	Dest = Dest >> Src	***Arithmetic right shift***
`shrq`	*Src,Dest*	Dest = Dest >> Src	***Logical right shift ()***
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

- Watch out for the argument order

Signed vs. Unsigned

- No distinction between signed and unsigned data in the operations (why?)
- Operations have similar bit level behavior
  - Difference appears in interpretation of outcome
  - E.g., 5 [0101] + 5 [0101] = 10 [01010]
    - Using 4 bits: 10[1010] with unsigned vs. -6 [1010] with signed
    - Operation is same but interpretation of outcome differs
    - Interpretation depends on the need
- Only right shifting requires instructions that differentiate between signed vs. unsigned data

Some Arithmetic Operations

- One Operand Instructions
  - `incq` *Dest* *Dest = Dest + 1*
  - `decq` *Dest* *Dest = Dest – 1*
  - `negq` *Dest* *Dest = – Dest*
  - `notq` *Dest* *Dest = ~Dest*

Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - But, only used once

Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%rax`	`t1`, `t2`, `rval`
`%rdx`	`t4`
`%rcx`	`t5`