CS201 – Lecture 6
IA32 Data Access and Operations Part I

RAOUL RIVAS

PORTLAND STATE UINIVERSITY

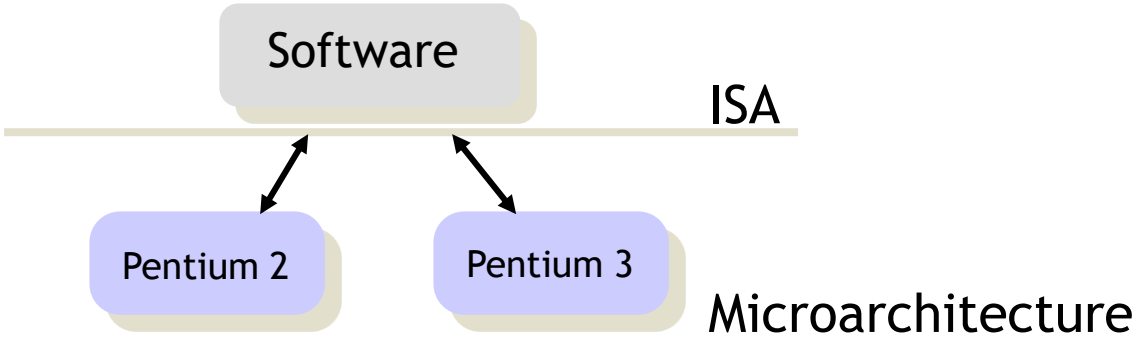
Announcements

Definitions

- Instruction Set Architecture: Processor Interface to programmers
  - Examples: instruction set specification, registers.
- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- Code Forms:
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code
  - Object Code: Machine code inside and Object File
- Example ISAs:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones

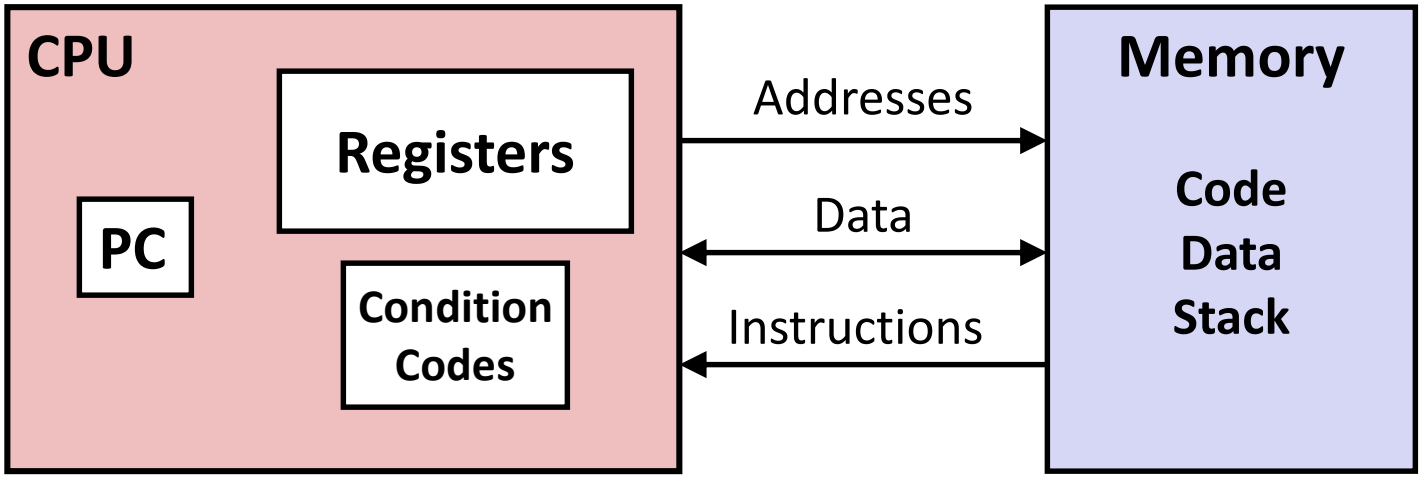
Instruction Set Architecture

- The ISA is an abstraction layer between hardware and software
  - Software doesn't need to know how the processor is implemented
  - Processors that implement the same ISA appears equivalent



- An ISA enables processor innovation without changing software
- Before ISAs, software was re-written/re-compiled for each new machine

ISA Overview



Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - 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

History of ISAs

- 1964: IBM System/360, the first computer family
  - IBM wanted to sell a range of machines that ran the same software
- 1960's, 1970's: Complex Instruction Set Computer (CISC) era
  - Much assembly programming, compiler technology immature
  - Hard to optimize, guarantee correctness, teach
- 1980's: Reduced Instruction Set Computer (RISC) era
  - Most programming in high-level languages, mature compilers
  - Simpler, cleaner ISA's facilitated pipelining, high clock frequencies
- 1990's: Post-RISC era
  - ISA compatibility outweighs any RISC advantage in general purpose
  - CISC and RISC chips use same techniques (pipelining, superscalar, ..)
  - Embedded processors prefer RISC for lower power, cost
- 2000's: Multi-core era

CISC vs RISC

- RISC has less number of instructions than CISC
- RISC instructions are simpler
  - Require many instruction to achieve the same as CISC
- RISC limits the instructions to register to register operations
- CISC might provide memory to register operations
- CISC executables tend to be smaller in size
  - Less instructions leads to smaller program sizes
- Intel IA32 is a CISC architecture and ARM is RISC
- Both ISA types can provide the same performance and possibly power usage!

ARM

intel

Intel x86 Processors

- Evolutionary design
- Backwards compatible up until 8086, introduced in 1978
- Added more features as time goes on
- Complex instruction set computer (CISC)
  - Addition in BCD format
  - Byte Swap
  - Increment, Decrement

Evolution of Intel Processors

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			

Assembly Overview

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

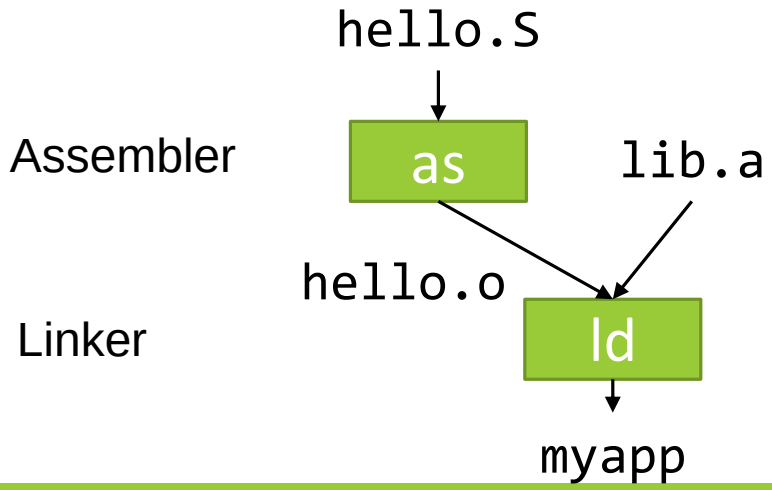
```
sumstore:
    pushq   %rbx
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    ret
```

- Very low level representation
- Specific to a particular ISA
- Lowest level we can program the processor

Interfacing with C

```
    .global main                # This is a global symbol hello.S
    .text                       # Start of text segment
main:
    mov  $message, %rdi
    call    puts
    ret                         # Return to C library code

    .data                       # Start of data segment
message:
    .asciz "Hello, world"       # asciz puts a 0 byte at the end
```




```
as --64 hello.S –o hello.o
gcc –o myapp hello.o
```

The Processor Manual

- The ISA is described in the Developer's Manual of the specific processor you are programming
  - It is the ultimate assembly reference!

You will probably use the online version of this manual for Homework 3

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

Assembly Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
  - Data values (Byte, Word, Doubleword, Quadword)
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Typical Assembly 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

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]**

- Machine Code
  - 3-byte instruction
  - Stored at address **0x40059e**

Dissasembling Object Files

Disassembled

```
0000000000400595 <sumstore>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3                   retq
```

- Disassembler

  `objdump -d sum`

  - Useful tool for examining machine 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

Disassembling in GDB

Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- **Within gdb Debugger**

  `gdb sum`

  `disassemble sumstore`

  - Disassemble procedure

  `x/14xb sumstore`

  - Examine the 14 bytes starting at `sumstore`

Registers

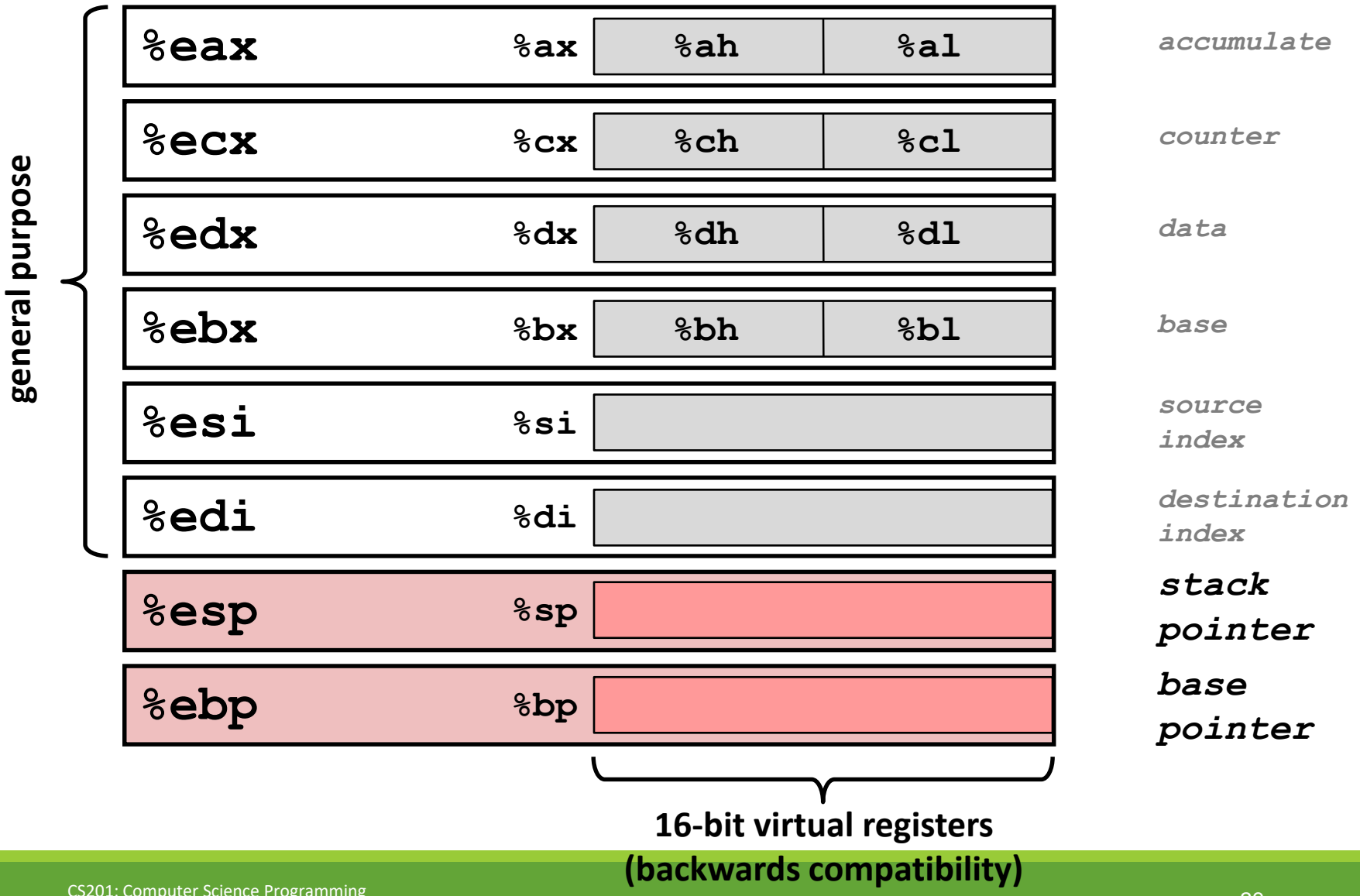
- Registers are tiny memory inside the processor used to store and operate over data.
  - Think of it as the “variables” of assembly
  - Most instructions will operate over registers
- They are very fast but the number of them is limited!

IA64 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)

IA-32 Registers

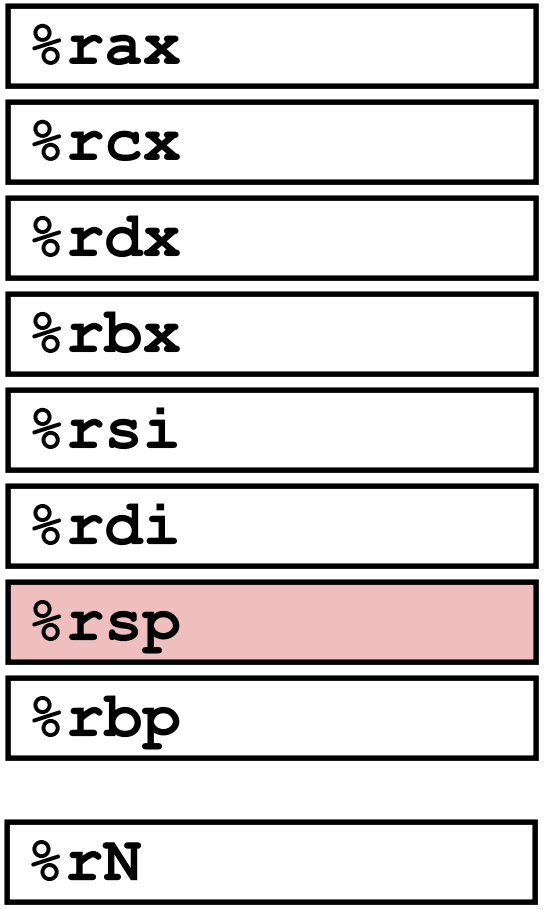


Copying Data

- Copying 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 16 integer registers
    - Example: **`%rax`, `%r13`**
    - But **`%rsp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **`(%rax)`**
    - Various other “address modes”



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 (Structures)

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

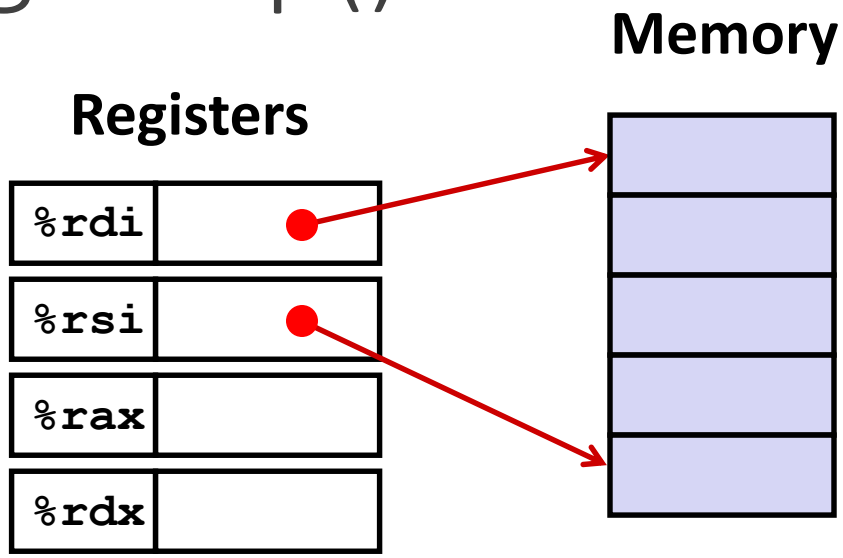
Swap() in Assembly

```
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
```

Additional Memory Addresing

- Most General Form

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

  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- Special Cases

  (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]]

Addressing 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`

Summary

- ISA is an abstraction layer between the microarchitecture and software
- CISC and RISC are two ISA paradigms with different advantages and disadvantages
- Disassemblers are used to convert Machine code to Assembly code
- Registers are small very fast memory inside the processor to perform operations
- MOV is a instruction used to copy data
- IA32 has several memory addressing modes