CMPT 295

Unit - Machine-Level Programming

Lecture 9 – Assembly language basics: Data, `move` operation

Last Lecture

- Review: von Neumann architecture
  - Data and code are both stored in memory during program execution

1. Question: How does our C program end up being represented as a series of 0's and 1's (i.e., as machine code)?
   - Compiler: C program -> assembly code -> machine level code
   - gcc: 1) C preprocessor, 2) C compiler, 3) assembler, 4) linker
2. Question: How does our C program (once it is represented as a series of 0's and 1's) end up being stored in memory? Loader
   - When C program is executed (e.g. from our demo: `./ss 5 6`)
3. Question: How does our C program (once it is represented as a series of 0's and 1's and it is stored in memory) end up being executed by the microprocessor (CPU)?
   - CPU executes C program by looping through the fetch-execute cycle

Summary - Turning C into machine level code - gcc

The Big Picture

→ Lab1

C program (.c)
→ sum_store.c

Preprocessed Source
→ sum_store.i

C Preprocessor `gcc -E sum_store.c > sum_store.i` (*text* → *text*)

Expands the header(s) found at the top of the C program by including their content into this C program

C Compiler `gcc -Og -S sum_store.i`
OR `gcc -Og -S sum_store.c`

Compiles the C program into an assembly language program

Assembly program (.s)
→ sum_store.s

Assembler `gcc -g -c main.s sum_store.s` (*text*, *text*)

Assembles the assembly language program into an object file (series of 0's and 1') which is a combination of machine language instructions, data and info needed to place all of this properly into memory

Object (.o)
→ sum_store.o

Executable
→ ss

Linker `gcc -o ss main.o sum_store.o` (*binary*, *binary*)

Combines independently assembled machine language programs with library routines into an executable file

Loader `./ss 5 6`

Loads machine code (from files) into proper memory locations for execution by CPU

optional

Disassembled object file
text & binary

← **Disassembler**
`objdump -d ss`
gdb/ddd debugger

ISA - Instruction Set Architecture

Computer executes it

CPU | Memory

Today's Menu

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmov*`
- Loops
- Function call – Stack
- Array
- Buffer Overflow
- Floating-point operations

Programming in C versus in x86-64 assembly language

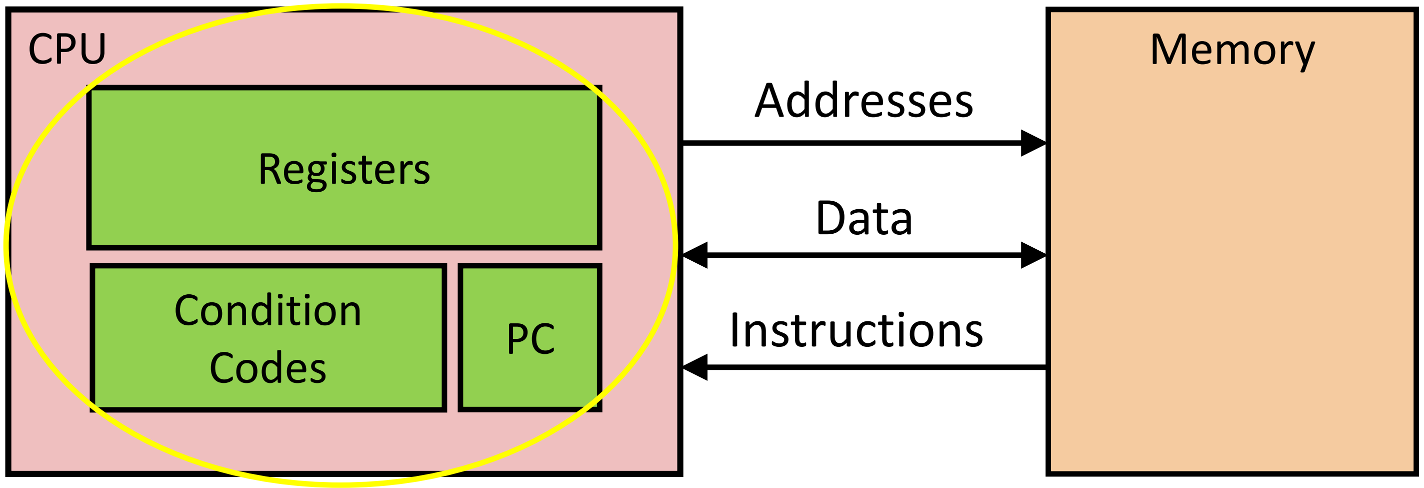
When programming in C, we can …

- Store/retrieve data into/from memory, i.e. variables
- Perform calculations on data
  - e.g., arithmetic, logic, shift
- Transfer control: decide what part of the program to execute next based on some condition
  - e.g., if-else, loop, function call

When programming in assembly language, we can do the same things, however …

Programming in x86-64 assembly

- … with assembly language (and machine code), parts of the microprocessor state are visible to assembly programmers that normally are hidden from C programmers
- As assembly programmers, we now have access to …



Hum … Why are we learning assembly language?

x86-64 Assembly Language - Data

- **Integral** numbers not stored in variables but in **registers**
  - Distinction between different integer size: 1, 2, 4 and 8 bytes
- Addresses not stored in pointer variables but in **registers**
  - Size: 8 bytes
  - Treated as integral numbers
- **Floating point numbers** stored in different **registers** than integral values
  - Distinction between different floating point numbers: 4 and 8 bytes
- No aggregate types such as arrays or structures

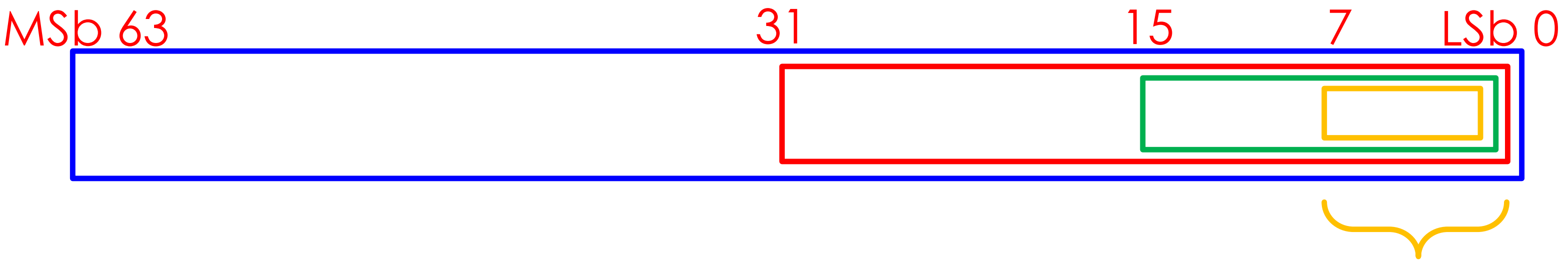
x86-64 Assembly Language – Data

Integer Registers

64-bit (quad)	32-bit (double)	16-bit (word)	8-bit (byte)	
63..0	31..0	15..0		7..0
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

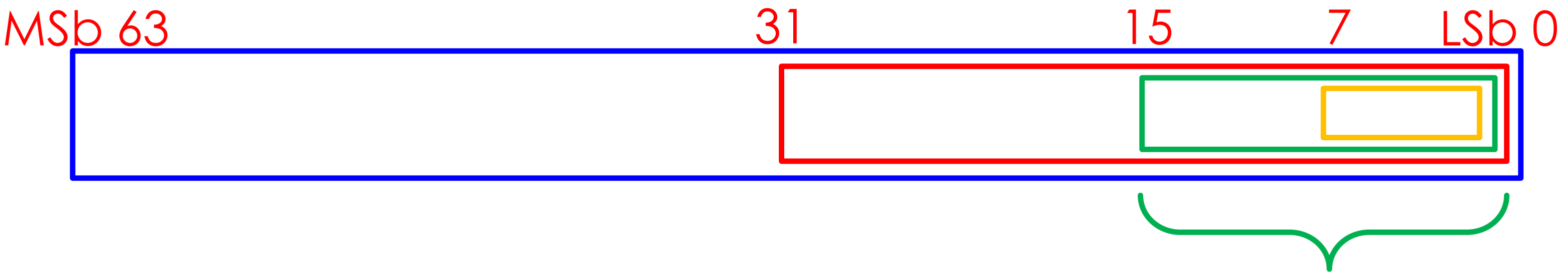
- Storage locations in CPU -> fastest storage
- 16 registers are used explicitly – must name them in assembly code %rax
- Some registers are used implicitly
  - e.g., PC, FLAGS
- Each register is 64 bits in size, but we can refer to its:
  - first byte LSB (8 bits),
  - first 2 bytes (16 bits),
  - first 4 bytes (32 bits),
  - or to all of its 8 bytes (64 bits)

About these integer registers!



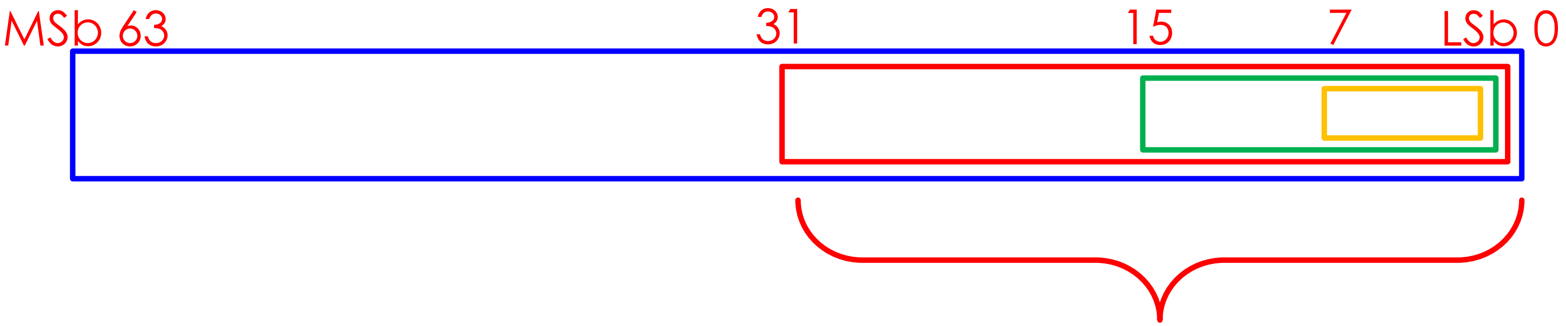
If I want 8 bits worth of data, then I can use register names such as `%al` or `%dil` or `%r12b`

About these integer registers!



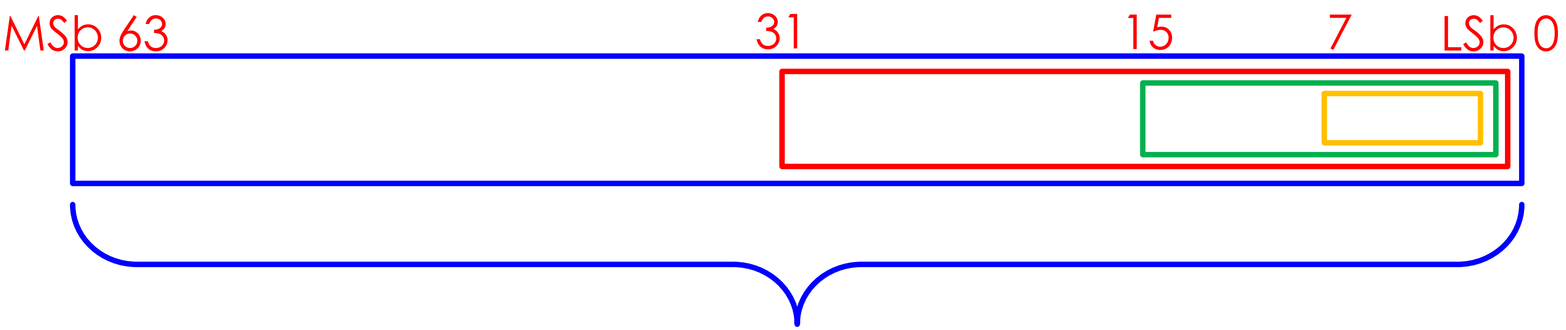
If I want **16** bits worth of data, then I can use register names such as `%ax` or `%di` or `%r12w`

About these integer registers!



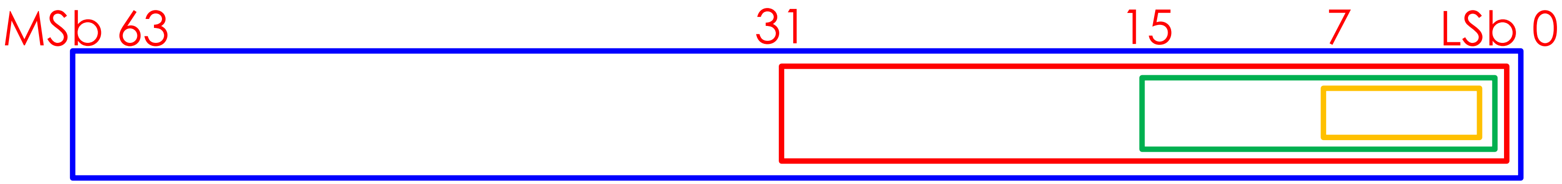
If I want **32** bits worth of data, then I can use register names such as `%eax` or `%edi` or `%r12d`

About these integer registers!



If I want **64** bits worth of data, then I can use register names such as `%rax` or `%rdi` or `%r12`

About these integer registers!



If I want **8** bits worth of data, then I can use register names such as `%al` or `%dil` or `%r12b`

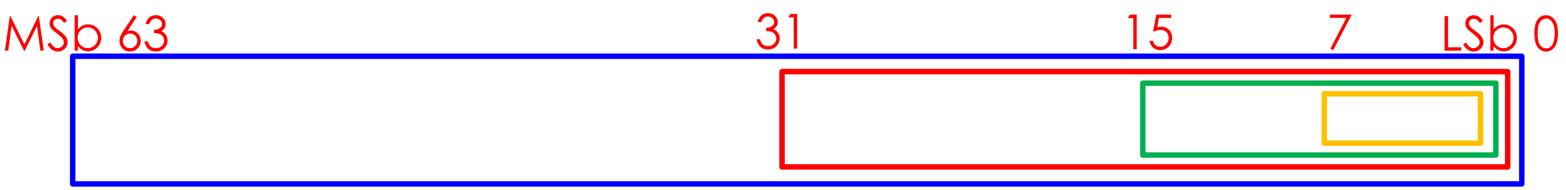
If I want **16** bits worth of data, then I can use register names such as `%ax` or `%di` or `%r12w`

If I want **32** bits worth of data, then I can use register names such as `%eax` or `%edi` or `%r12d`

If I want **64** bits worth of data, then I can use register names such as `%rax` or `%rdi` or `%r12`

Remember that for all 16 registers ...

Let's use the register associated with the names `%rax`, `%eax`, `%ax` and `%al` as an example:



- `%rax`, `%eax`, `%ax` and `%al` all refer to the **same register**
- However…
  - Each refer to a different section of this register
  - `%rax` refers to all 64 bits of this register
  - `%eax` refers to only 32 bits of this register
    - the LS 32 bits of it -> bit 0 to bit 31
  - `%ax` refers to only 16 bits of this register
    - the LS 16 bits of it -> bit 0 to bit 15
  - `%al` refers to only 8 bits of this register
    - the LS 8 bits of it -> bit 0 to bit 7

x86-64 Assembly Language - Instructions

- "2 operand" assembly language
- x86-64 functionally complete -> i.e., it is "Turing complete"
  - 3 classes of instructions

1. Memory reference => Data transfer instructions
   - Transfer data between memory and registers
     - **Load** data from memory into register
     - **Store** register data into memory
     - **Move** data from one register to another
2. Arithmetic and logical => Data manipulation instructions
   - Perform calculations on register data
     - e.g., arithmetic, logic, shift
3. Branch and jump => Program control instructions
   - Transfer control
     - Unconditional jumps to/from functions
     - Unconditional/conditional branches

Homework: movl $0xFF4150AC, %eax

32 bits   32 bits

Move data – mov*

1. Memory reference => Data transfer instructions

- Transfer data between memory and registers
- Syntax: **mov*** ***Source, Destination***
- Example: **movq %rdi, %rax**

- Allowed moves:
  - From register to register (**Move**)
  - From memory to register (**Load**)
  - From register to memory (**Store**)

- Conditional move (**cmov***)
  - Same as above, but based on result of comparison

* -> Size designator

q	->	long	64
l	->	int	32
w	->	short	16
b	->	char	8

Demo – Swap Function

- Problem: Let's swap the contents of two variables
- For now, we need to know that
  - Argument 1 of function swap(…) -> saved in `%rdi`
  - Argument 2 of function swap(…) -> saved in `%rsi`

Demo – Swap Function

```c
void swap(long *xp, long *yp)
{
  long L1 = *xp;
  long L2 = *yp;
  *xp = L2;
  *yp = L1;
  return;
}
```

because contains an address → 64 bits ∴ %rdi & %rsi are used to hold the value of xp & yp

xp → %rdi
yp → %rsi

```
swap:
    movq    (%rdi), %rax      # L1 = *xp
    movq    (%rsi), %rdx      # L2 = *yp
    movq    %rdx, (%rdi)      # *xp = L2
    movq    %rax, (%rsi)      # *yp = L1
    ret
```

indirect

comments

Registers

%rdi	0x0020
%rsi	0x0000
%rax	123
%rdx	456

Memory

Address	
0x0020	~~123~~ 456
0x0018	
0x0010	
0x0008	
0x0000	~~456~~ 123

=> Remember:

Compressed view of memory

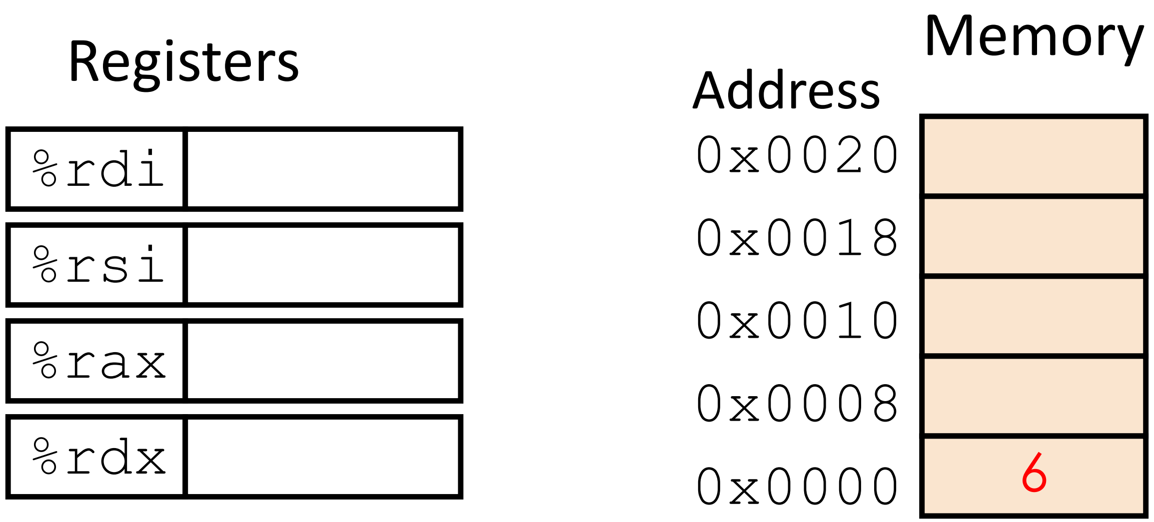
Operand Combinations for `mov*`

	Source	Dest	Src, Dest (Memory addressing modes)	in C
`mov*`	*Immediate*	*Register*	`movq $0x4,%rax`	`result = 0x4;`
		Memory	`movq $-147,(%rax)`	`*result = -147;`
	Register	*Register*	`movq %rax,%rdx`	`var1 = result;`
		Memory	`movq %rax,(%rdx)`	`*var1 = result;`
	Memory	*Register*	`movq (%rax),%rdx`	`var1 = *result;`

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

Homework 2

- Since we cannot do memory-memory transfer with a single `mov*` instruction …
  - Can you write a little x86-64 assembly program that transfers data stored at address `0x0000` to address `0x0018` ?



Summary

- As x86-64 assembly s/w dev., we now get to see more of the microprocessor (CPU) state: PC, registers, condition codes
- x86-64 assembly language – ***Data***
  - 16 integer registers of 1, 2, 4 or 8 bytes + memory address of 8 bytes
  - Floating point registers of 4 or 8 bytes
  - No aggregate types such as arrays or structures
- x86-64 assembly language – ***Instructions***
  - `mov*` instruction family
    - From register to register
    - From memory to register
    - From register to memory
  - Memory addressing modes
  - *Cannot do memory-memory transfer with a single mov* instruction*

Next Lecture

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmov*`
- Loops
- Function call – Stack
- Array
- Buffer Overflow
- Floating-point operations