Computer Architecture
CS-211

Spring 2017 | Recitation
Abu Shoeb

Agenda

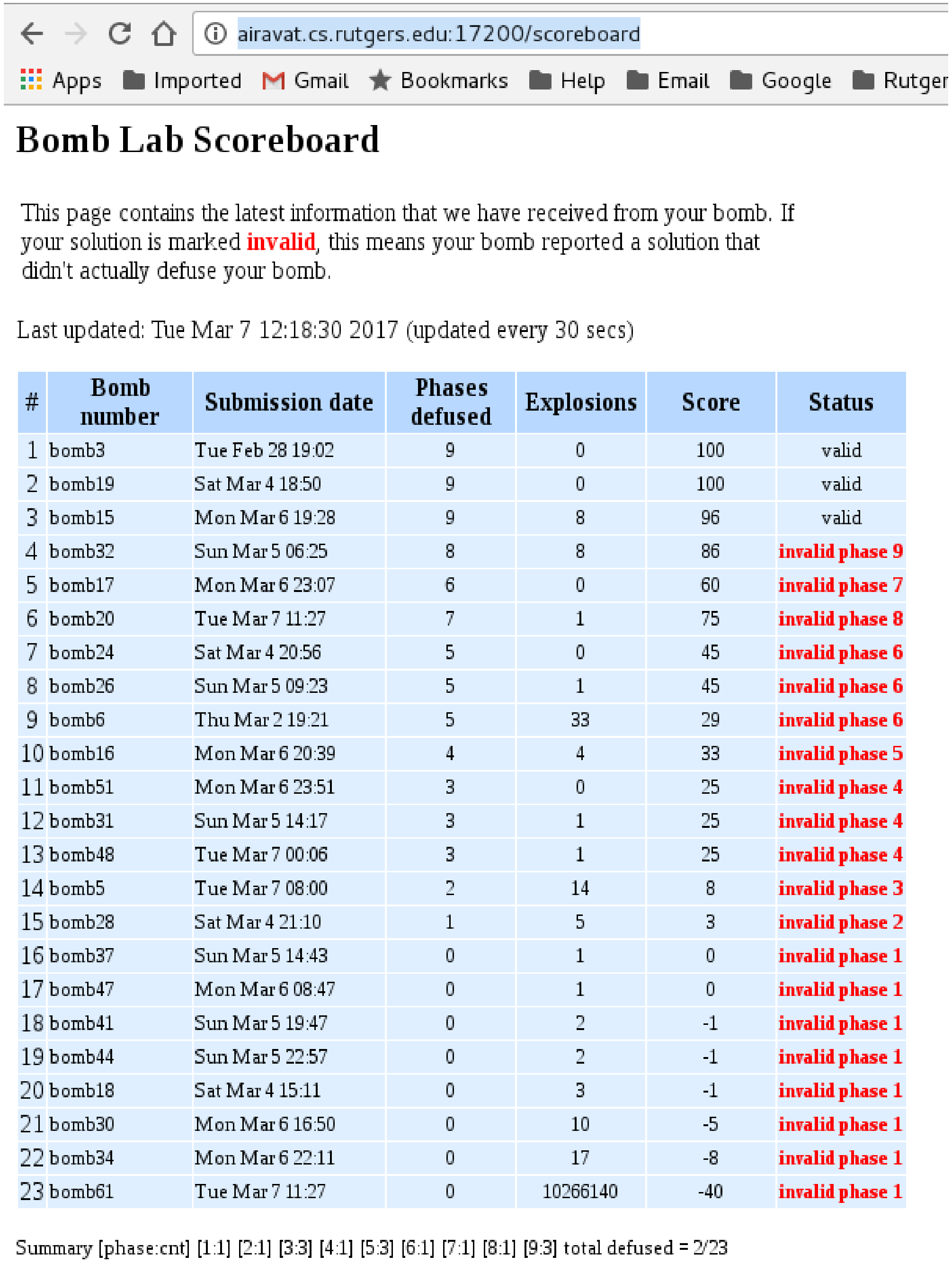
- Programming Assignment 3 (Binary Bomb Lab)
  - Overview
  - How to defuse the bomb using GDB!
  - Some useful resources
- Assembly Language

PA 3 – Bomb Lab

- Download bomb<N>.tar (N represents your ID)
  - http://airavat.cs.rutgers.edu:17200
  - Don’t download more than 2 bombs!
  - Download this in iLab machines or copy downloaded bomb into iLab machines
- Untar your bomb
  - $ tar -xvf bomb<N>
  - bomb<N> directory will have
    - bomb, bomb.c, README
- Solve using GDB!
- See score at http://airavat.cs.rutgers.edu:17200/scoreboard
- Put your results/input in defuser.txt
- Submit your bomb along with defuser.txt

PA 3 – Scoreboard

- Remember : You will lose **0.5** points for each explodes!



Bomb Lab Scoreboard

This page contains the latest information that we have received from your bomb. If your solution is marked **invalid**, this means your bomb reported a solution that didn't actually defuse your bomb.

Last updated: Tue Mar 7 12:18:30 2017 (updated every 30 secs)

#	Bomb number	Submission date	Phases defused	Explosions	Score	Status
1	bomb3	Tue Feb 28 19:02	9	0	100	valid
2	bomb19	Sat Mar 4 18:50	9	0	100	valid
3	bomb15	Mon Mar 6 19:28	9	8	96	valid
4	bomb32	Sun Mar 5 06:25	8	8	86	**invalid phase 9**
5	bomb17	Mon Mar 6 23:07	6	0	60	**invalid phase 7**
6	bomb20	Tue Mar 7 11:27	7	1	75	**invalid phase 8**
7	bomb24	Sat Mar 4 20:56	5	0	45	**invalid phase 6**
8	bomb26	Sun Mar 5 09:23	5	1	45	**invalid phase 6**
9	bomb6	Thu Mar 2 19:21	5	33	29	**invalid phase 6**
10	bomb16	Mon Mar 6 20:39	4	4	33	**invalid phase 5**
11	bomb51	Mon Mar 6 23:51	3	0	25	**invalid phase 4**
12	bomb31	Sun Mar 5 14:17	3	1	25	**invalid phase 4**
13	bomb48	Tue Mar 7 00:06	3	1	25	**invalid phase 4**
14	bomb5	Tue Mar 7 08:00	2	14	8	**invalid phase 3**
15	bomb28	Sat Mar 4 21:10	1	5	3	**invalid phase 2**
16	bomb37	Sun Mar 5 14:43	0	1	0	**invalid phase 1**
17	bomb47	Mon Mar 6 08:47	0	1	0	**invalid phase 1**
18	bomb41	Sun Mar 5 19:47	0	2	-1	**invalid phase 1**
19	bomb44	Sun Mar 5 22:57	0	2	-1	**invalid phase 1**
20	bomb18	Sat Mar 4 15:11	0	3	-1	**invalid phase 1**
21	bomb30	Mon Mar 6 16:50	0	10	-5	**invalid phase 1**
22	bomb34	Mon Mar 6 22:11	0	17	-8	**invalid phase 1**
23	bomb61	Tue Mar 7 11:27	0	10266140	-40	**invalid phase 1**

Summary [phase:cnt] [1:1] [2:1] [3:3] [4:1] [5:3] [6:1] [7:1] [8:1] [9:3] total defused = 2/23

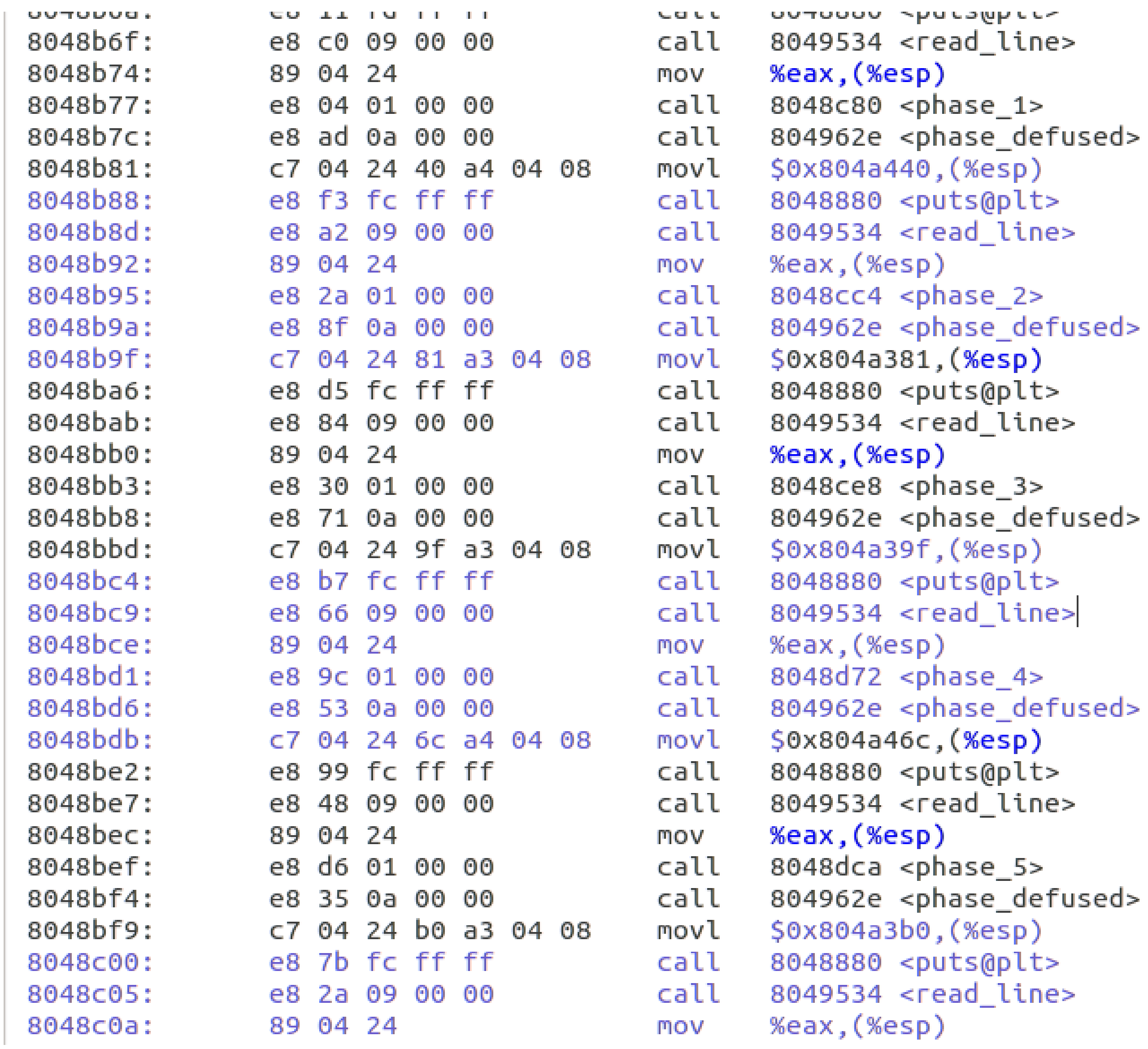
How to Defuse It!

- One way to do it by debugging using GDB
  - $ gdb bomb (run in gdb)
  - Set break point for each phase (e.g. (gdb) break phase_1) (this will help you not to explode the bomb)
  - Run the program ((gdb) run)

- Useful Commands for binary bomb
  - Print bomb’s symbol table ($ objdump -t bomb)
  - Disassemble the code ($ objdump -d bomb)
  - Display printable strings ($ strings -t x bomb)

- You can save output of commands into file
  - Example : $ objdump -d bomb > bomb-assembly.txt

How to Defuse It!



```
8048b6a:       e8 11 fd ff ff          call   8048880 <puts@plt>
8048b6f:       e8 c0 09 00 00          call   8049534 <read_line>
8048b74:       89 04 24                mov    %eax,(%esp)
8048b77:       e8 04 01 00 00          call   8048c80 <phase_1>
8048b7c:       e8 ad 0a 00 00          call   804962e <phase_defused>
8048b81:       c7 04 24 40 a4 04 08    movl   $0x804a440,(%esp)
8048b88:       e8 f3 fc ff ff          call   8048880 <puts@plt>
8048b8d:       e8 a2 09 00 00          call   8049534 <read_line>
8048b92:       89 04 24                mov    %eax,(%esp)
8048b95:       e8 2a 01 00 00          call   8048cc4 <phase_2>
8048b9a:       e8 8f 0a 00 00          call   804962e <phase_defused>
8048b9f:       c7 04 24 81 a3 04 08    movl   $0x804a381,(%esp)
8048ba6:       e8 d5 fc ff ff          call   8048880 <puts@plt>
8048bab:       e8 84 09 00 00          call   8049534 <read_line>
8048bb0:       89 04 24                mov    %eax,(%esp)
8048bb3:       e8 30 01 00 00          call   8048ce8 <phase_3>
8048bb8:       e8 71 0a 00 00          call   804962e <phase_defused>
8048bbd:       c7 04 24 9f a3 04 08    movl   $0x804a39f,(%esp)
8048bc4:       e8 b7 fc ff ff          call   8048880 <puts@plt>
8048bc9:       e8 66 09 00 00          call   8049534 <read_line>
8048bce:       89 04 24                mov    %eax,(%esp)
8048bd1:       e8 9c 01 00 00          call   8048d72 <phase_4>
8048bd6:       e8 53 0a 00 00          call   804962e <phase_defused>
8048bdb:       c7 04 24 6c a4 04 08    movl   $0x804a46c,(%esp)
8048be2:       e8 99 fc ff ff          call   8048880 <puts@plt>
8048be7:       e8 48 09 00 00          call   8049534 <read_line>
8048bec:       89 04 24                mov    %eax,(%esp)
8048bef:       e8 d6 01 00 00          call   8048dca <phase_5>
8048bf4:       e8 35 0a 00 00          call   804962e <phase_defused>
8048bf9:       c7 04 24 b0 a3 04 08    movl   $0x804a3b0,(%esp)
8048c00:       e8 7b fc ff ff          call   8048880 <puts@plt>
8048c05:       e8 2a 09 00 00          call   8049534 <read_line>
8048c0a:       89 04 24                mov    %eax,(%esp)
```

Some Useful GDB Commands

```
(gdb) ni  - next instruction
(gdb) si  - step in (e.g. step into function)
(gdb) step  - step out
(gdb) disas  - disassemble instructions
(gdb) until  *addr – jump to the given addr
(gdb) i r – print all reg values
(gdb) x/s addr – print value of the addr (similarly x/d)
```

GDB

- https://www.csee.umbc.edu/~cpatel2/links/310/nasm/gdb_help.shtml

SSH Tunnel with Firefox

Linux

https://ubuntuforums.org/showthread.php?t=723025

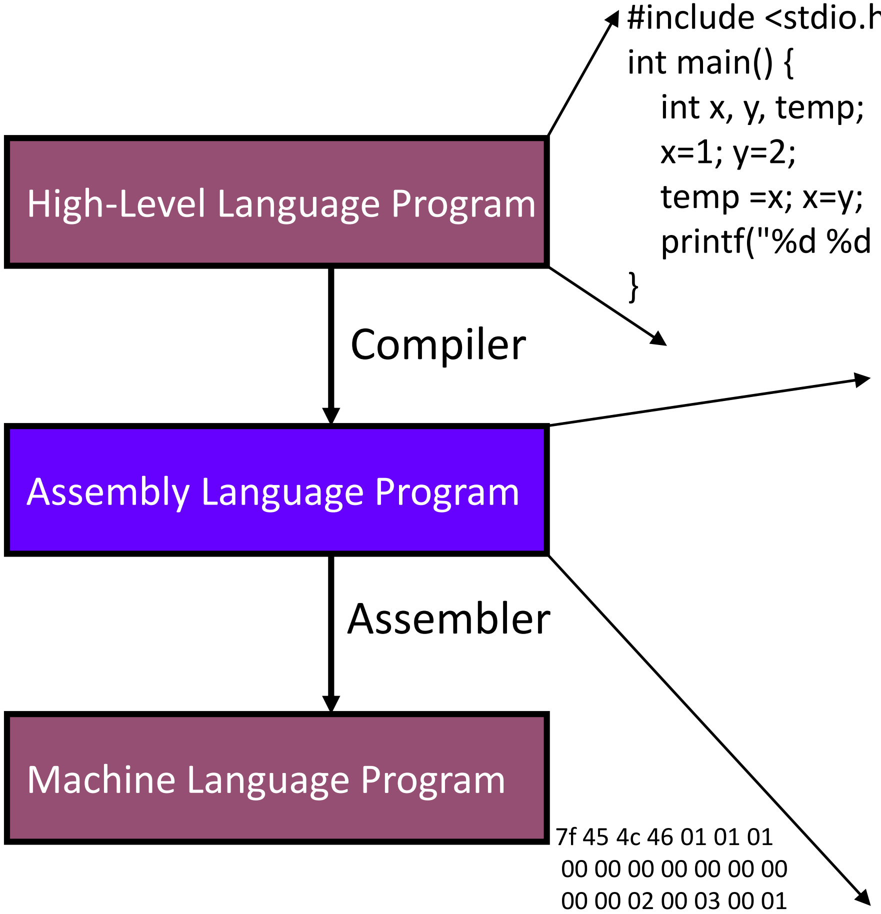
```
$ ssh -D 9999 -C netId@iLab
```

Windows

https://www.sotechdesign.com.au/browsing-the-web-through-a-ssh-tunnel-with-firefox-and-putty-windows/

Assembly Language

Programming Meets Hardware



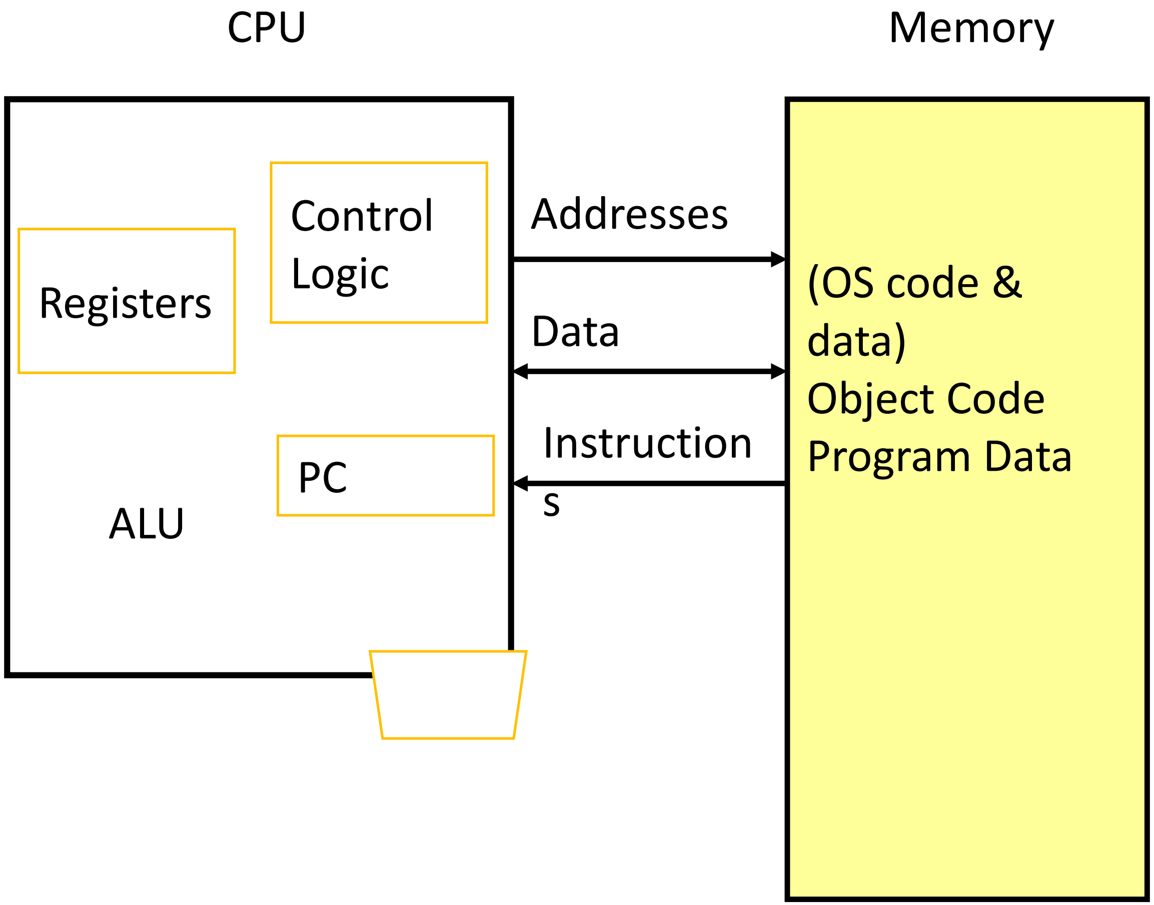
```
#include <stdio.h>
int main() {
   int x, y, temp;
   x=1; y=2;
   temp =x; x=y;  y=temp;
   printf("%d %d %d\n",x,y,temp);
}
```

```
movl        $1, -8(%ebp)
movl        $2, -12(%ebp)
movl        -8(%ebp), %eax
movl        %eax, -16(%ebp)
movl        -12(%ebp), %eax
movl        %eax, -8(%ebp)
movl        -16(%ebp), %eax
movl        %eax, -12(%ebp)
movl        -16(%ebp), %eax
movl        %eax, 12(%esp)
movl        -12(%ebp), %eax
movl        %eax, 8(%esp)
movl        -8(%ebp), %eax
movl        %eax, 4(%esp)
```

ISA

```
7f 45 4c 46 01 01 01
 00 00 00 00 00 00 00
 00 00 02 00 03 00 01
00 00 00 f0 82 04 08
34 00 00 00 c4 0c 00
00 00 00 00 00 34 00
```

Assembly Programmer’s View



CPU

Memory

Registers

Control Logic

PC

ALU

Addresses

Data

Instructions

(OS code & data)
Object Code
Program Data

Assembly Characteristics

- Primitive 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

Instruction Format

- General format:

  `opcode operands`
- Opcode:
  - Short mnemonic for instruction’s purpose
    - `movb,addl, etc.`
- Operands:
  - Immediate, register, or memory
  - Number of operands command-dependent
- Example:
  - movl %ebx, (%ecx)

MOV instruction

- Most common instruction is data transfer instruction
    - mov S, D
        - Copy value at S from D
- Used to copy data from:
    - Memory to register
    - Register to memory
    - Register to register
    - Constant to register

Data Formats

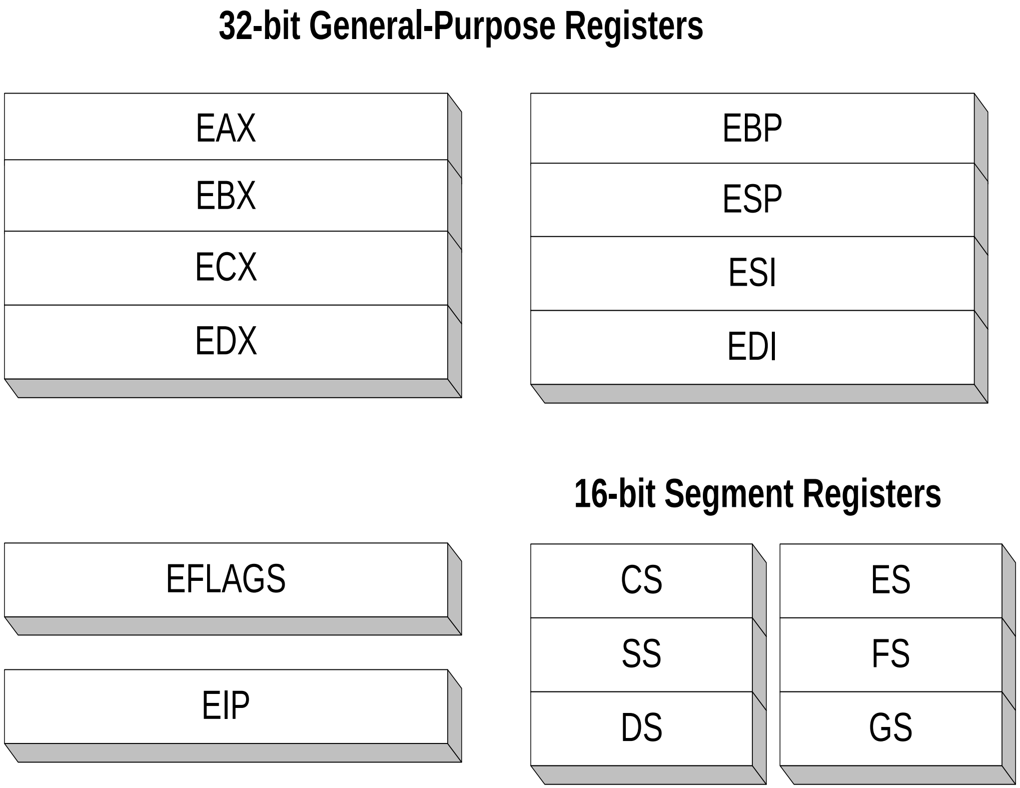
- Byte: 8 bits
  - E.g., char
- Word: 16 bits (2 bytes)
  - E.g., short int
- Double Word: 32 bits (4 bytes)
  - E.g., int, float
- Quad Word: 64 bits (8 bytes)
  - E.g., double
- Instructions can operate on any data size
  - **`movl, movw, movb`**
    - **`Move double word, word, byte, respectively`**
  - End character specifies what data size to be used

Registers

- Registers are CPU components that hold data and address
- Much faster to access than memory
- It is used to speed up CPU operations
- Categories
    - General registers
        - Data registers (Holds operands)
        - Pointer & index registers (Holds references to addresses as well as indices)
    - Control Register (e.g. CF,ZF)
    - Segment registers (Holds starting address of program segments)
        - CS, DS, SS, ES

Registers Overview

- Named storage locations inside the CPU, optimized for speed



32-bit General-Purpose Registers

EAX
EBX
ECX
EDX

EBP
ESP
ESI
EDI

EFLAGS

EIP

16-bit Segment Registers

CS
SS
DS

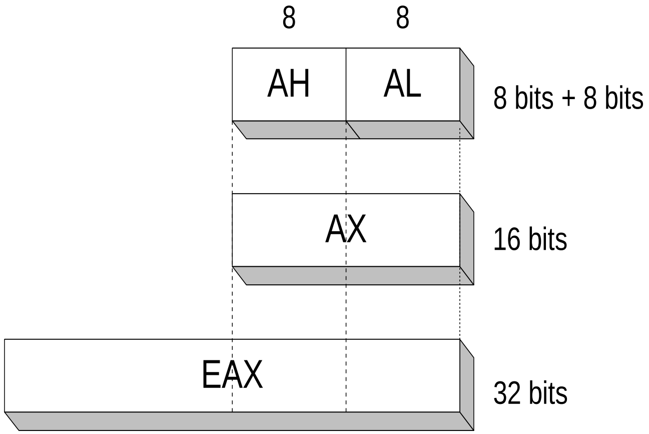
ES
FS
GS

Data Registers 1

- AX is the primary accumulator
  - Used in most arithmetic instruction
- BX is the base register
  - Could be used in indexed addressing
- CX is the count register
  - Store the loop count in iterative operations
- DX is the data register
  - Used in input / output operations

Data Registers 2

Can use 8-bit, 16-bit, or 32-bit name



32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Pointer Registers

- ESP is stack pointer
  - It refers to be current position of data or address within the program stack
  - Changed by push, pop instructions
- EBP is frame pointer
  - Referencing the parameter variables passed to a subroutine
- EIP is instruction pointer
  - It stores the offset address of the next instruction to be executed

Control Registers

- Overflow flag (OF)
  - Indicates the overflow of a high-order bit
- Carry flag (CF)
  - Contains the carry of 0 or 1 from high-order bit after arithmetic operation
  - Stores the last bit of a shift or rotate operation
- Sign flag (SF)
  - Shows the sign of the result of an arithmetic operation
  - Positive -> 0, Negative -> 1
- Zero Flag (ZF)

Segment Registers

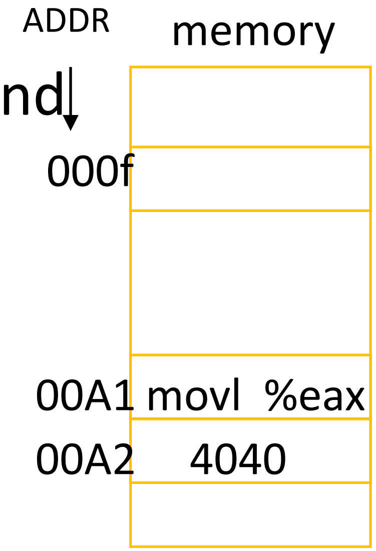
- Segments are specific areas defined in a program for containing data, code, and stack
- Code segment
  - Contains the instructions to be executed
- Data segment
  - Contains data, constants and work areas
- Stack segment
  - Contains data and return addresses of procedures

Labels

- Act as place markers
  - Marks the address of code and data (can be used to represent an address)
- Data label
  - Must be unique
  - Ex) myArray (not followed by colon)
- Code label
  - Target of jump or loop instructions
  - Ex) L1: (followed by colon)

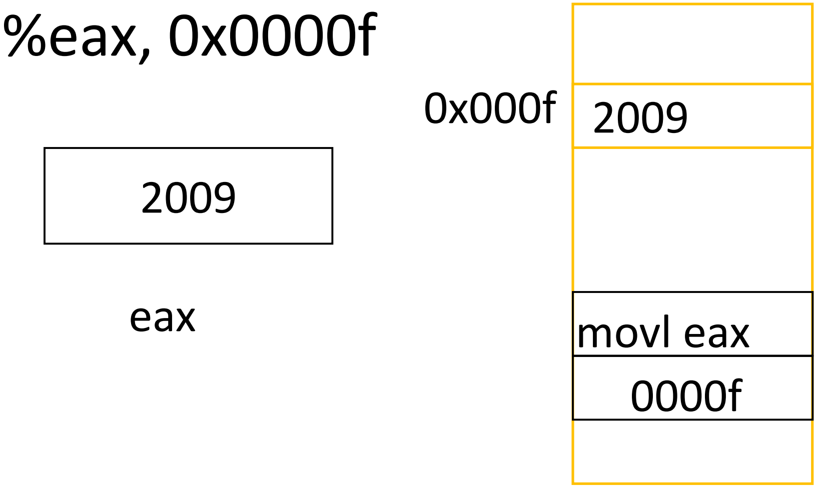
Immediate Addressing

- Operand is immediate
  - Operand value is found immediately following the instruction
  - $ in front of immediate operand
  - E.g., movl $0x4040, %eax



Direct Addressing

- Address of operand is found immediately after the instruction
  - Also known as direct addressing or absolute address
  - movl %eax, 0x0000f



Register Mode Addressing

- Use % to denote register
  - E.g., %eax
- Source operand: use value in specified register
- Destination operand: use register as destination for value
- Examples:
  - movl %eax, %ebx
    - Copy content of %eax to %ebx
  - movl $0x4040, %eax (immediate addressing)
    - Copy 0x4040 to %eax
  - movl %eax, 0x0000f (direct addressing)
    - Copy content of %eax to memory location 0x0000f

Indirect Mode Addressing

- Content of operand is an address
  - Designated as parenthesis around operand
- Offset can be specified as immediate mode
- Examples:
  - movl (%ebp), %eax
    - Copy value from memory location whose address is in ebp into eax
  - movl -4(%ebp), %eax
    - Copy value from memory location whose address is -4 away from content of ebp into eax

Indexed Mode Addressing

- Add content of two registers to get address of operand
  - movl (%eab, %esi), %eax
    - Copy value at (address = eab + esi) into eax
- Useful for dealing with arrays
  - If you need to walk through the elements of an array
  - Use one register to hold base address, one to hold index
    - E.g., implement C array access in a for loop

Thanks!

Any questions?