CS240 Laboratory 7
More on Pointers/Introduction to Disassembly

MYSTERY0.C

```
#include <stdlib.h>
#include <stdio.h>
#include "hexdump.h"

 void mystery0(char* ps, char* pa) {
  while (*ps) {
    *pa++ = *ps++;
  }
  *pa = '\0';
}

 int main() {
   char a[7] = {'h', 'e', 'l', 'l', 'o', '!', '\0'};
   char b[7];    // hexdump(b, 16);
   printf("\n");
   mystery0(a, b);   // hexdump(b, 16);
   printf("a = \"%s\"\n", a);
   printf("b = \"%s\"\n\n", b);
   mystery0("cs 240", a);   // hexdump(b, 16);
   printf("a = \"%s\"\n", a);
  printf("b = \"%s\"\n\n", b);
  mystery0("0xF", &a[2]);   // hexdump(b, 16);
  printf("a = \"%s\"\n", a);
  printf("b = \"%s\"\n\n", b);
}
```

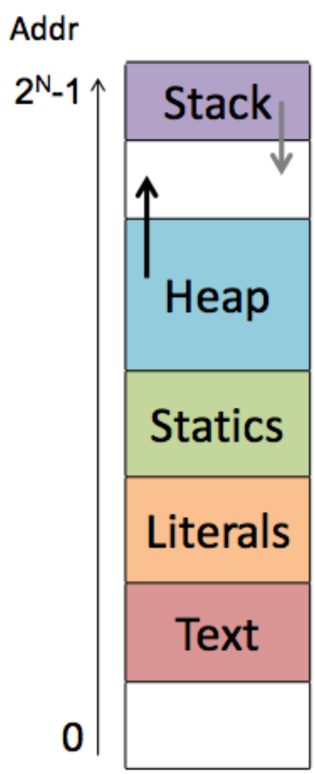
MYSTERY1.C

```c
#include <stdlib.h>
#include <stdio.h>
#include "hexdump.h"

void copy(char* src, char* dst) {
   while (*src) {
     *dst++ = *src++;
   }
   *dst = '\0';
}

int main() {
   char* p;
   char a[8];
   int x = 19;
    char b[4];
    p = &a[4];    // hexdump(b, 32);
   printf("\n");
   copy("Hello!", a);
   copy("Hi!", b);   // hexdump(b, 32);
   printf("x = %d\n", x);
   printf("p = %p\n", p);
   printf("a = \"%s\"\n", a);
   printf("b = \"%s\"\n\n", b);
   copy("Hi, CS 240!", b);   // hexdump(b, 32);
   printf("x = %d\n", x);
   printf("p = %p\n", p);
   printf("a = \"%s\"\n", a);
   printf("b = \"%s\"\n\n", b);
   copy("What happens if we use a long string?", b);   // hexdump(b, 64);
   printf("x = %d\n", x);   printf("p = %p\n", p);
   printf("a = \"%s\"\n", a);   printf("b = \"%s\"\n\n", b);
   copy("Hi?", p);   // hexdump(b, 64);
   printf("x = %d\n", x);   printf("p = %p\n", p);
   printf("a = \"%s\"\n", a);
   printf("b = \"%s\"\n\n", b);
}
```

Memory Layout



Perm	Contents	Managed by	Initialized
RW	Procedure context	Compiler	Run-time
RW	Dynamic data structures	Programmer, malloc/free, new/GC	Run-time
RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
R	String literals	Compiler/ Assembler/Linker	Startup
X	Instructions	Compiler/ Assembler/Linker	Startup

Segmentation Fault accessing address outside legal area of memory

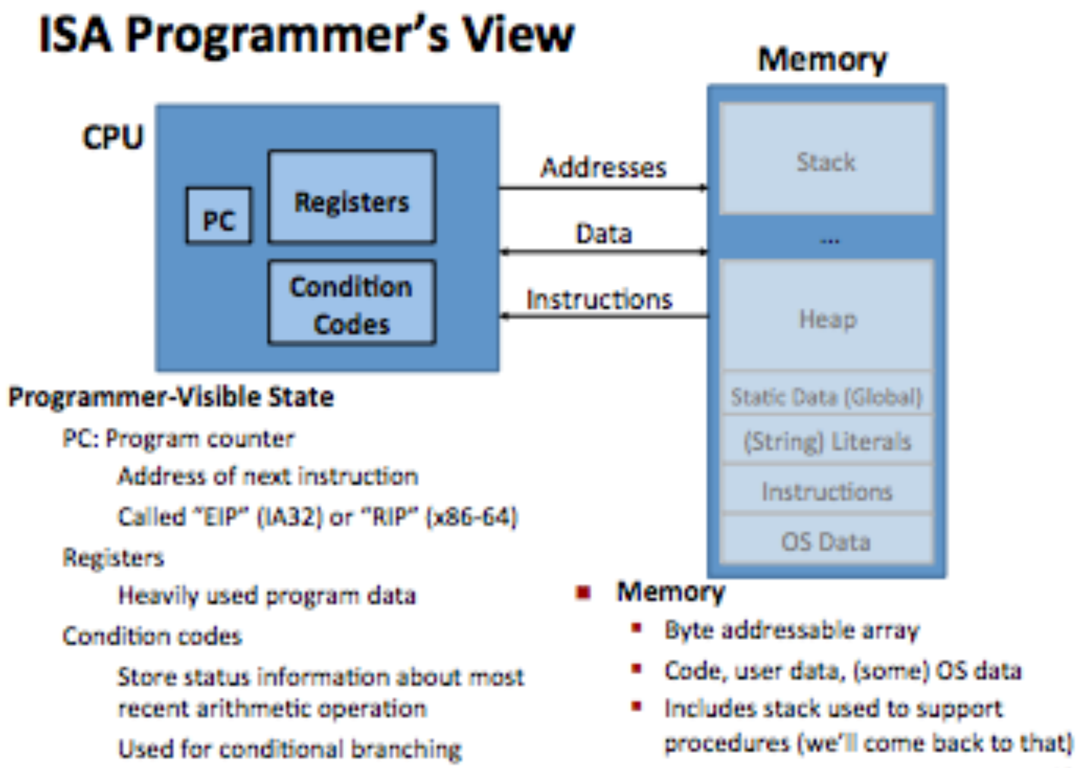
Bus Error accessing misaligned or other problematic address

Instruction Set Architecture (ISA)

The ISA defines:

- system state (e.g. registers, memory, program counter)

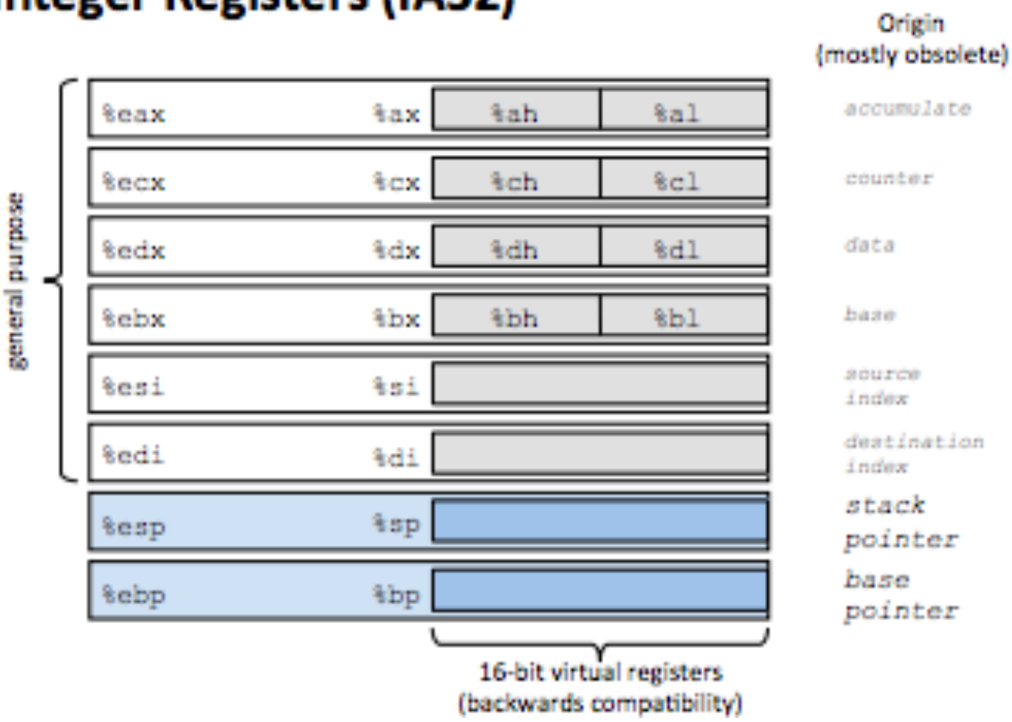
- instructions the CPU can execute
- effect of each instruction on system state



Condition Code Flags

CF, ZF,SF,OF (Carry, Zero,Sign, Overflow – set by arithmetic operations)

Integer Registers (IA32)



Three Basic Kinds of Instructions

Transfer data between memory and register

Load data from memory into register

%reg = Mem[address]

Store register data into memory

Mem[address] = %reg

> Remember:
> memory is indexed
> just like an array[]
> of bytes!

Perform arithmetic function on register or memory data

c = a + b; z = x << y; i = h & g;

Transfer control: what instruction to execute next

Unconditional jumps to/from procedures

Conditional branches

Operand Types

Immediate:

$0x400, $-533

Register:

%eax, %edx

Memory:

- indirect: *(%eax)*
– displacement: *8(%eax)*

Complete Memory Addressing Modes

Memory addresses used by **mov** (and other) instructions can be computed in several different ways.

Most General Form:

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

D: Constant "displacement" value represented in 1, 2, or 4 bytes

Rb: Base register: Any register

Ri: Index register: Any except **%esp** (or **%rsp** if 64-bit) ; **%ebp** unlikely

S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Special Cases: can use any combination of D, Rb, Ri and S

(Rb,Ri)	**Mem[Reg[Rb]+Reg[Ri]]**	(S=1, D=0)
D(Rb,Ri)	**Mem[Reg[Rb]+Reg[Ri]+D]**	(S=1)
(Rb,Ri,S)	**Mem[Reg[Rb]+S*Reg[Ri]]**	(D=0)

X86 Instructions

Moving Data

movl Src,Dest

Load Effective Address - compute address or arithmetic expression of the form x + k*i
(does not set the condition flags!)

leal Src,Dest

Arithmetic/Logical operations – 2 operands

addl Src,Dest
subl Src,Dest
imull Src,Dest
shrl Src,Dest
sarl Src, Dest
shll Src,Dest
sall Src, Dest
shrl Src,Dest

xorl Src,Dest
andl Src,Dest
orl Src,Dest

mull Src,Dest
imull Src,Dest
divl Src,Dest
idivl Src,Dest

Arithmetic/Logical operations – 1 operand

incl Dest
decl Dest
negl Dest
notl Dest

Zero Extend from Byte to Long Word

movzbl Src,Dest

Setting Condition Codes Explicitly

cmpl/cmpq Src2,Src1	sets flags based on value of Src2 – Src1, discards result
testl/testq Src2,Src1	sets flags based on a & b, discards result useful to have one of the operands be a mask

Inspecting Condition Codes

SetX Instructions

Set a single byte to 0 or 1 based on combinations of condition codes

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Control Flow

Conditional jump instructions) in X86 implement the following high-level constructs:

- if (condition) then {...} else {…}
- while (condition) {…}
- do {…} while (condition)
- for (initialization; condition; iterative) {...}

Unconditional jumps are used for high-level constructs such as:

- break
- continue

Jumping

jX Instructions

Jump to different part of code depending on condition codes

Takes address as argument

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

PC-relative Addressing

Jump instructions encode the offset from next instruction to destination PC, instead of the absolute address of the destination (makes it easier to relocate the code)

Translation of C program to X86

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

```
swap:
   pushl %ebp               } Set
   movl  %esp,%ebp          } Up
   pushl %ebx

   movl 12(%ebp),%ecx       }
   movl 8(%ebp),%edx        }
   movl (%ecx),%eax         } Body
   movl (%edx),%ebx         }
   movl %eax,(%edx)         }
   movl %ebx,(%ecx)         }

   movl -4(%ebp),%ebx       }
   movl %ebp,%esp           } Finish
   popl %ebp                }
   ret                      }
```

- Instructions can be in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Compiler optimization can do some surprising things!

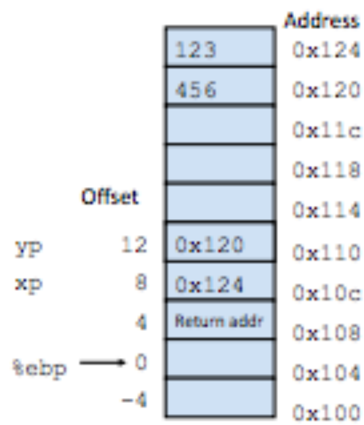
Register to Variable mapping

%ecx = yp
%edx = xp
%eax = t1
%ebx = t0

Register Values

%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Stack Contents



Object Code

Code for sum

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address `0x401040`
- Not at all obvious where each instruction starts and ends

Assembler

Translates `.s` into `.o`

Binary encoding of each instruction

Nearly-complete image of executable code

Missing links between code in different files

Linker

Resolves references between object files and (re)locates their data

Combines with static run-time libraries

e.g., code for `malloc`, `printf`

Some libraries are *dynamically linked* when program begins execution

Disassembly

Tools can be used to examine bytes of object code (executable program) and reconstruct the assembly source .

objdump

$ *objdump –t p*

Prints out the program's symbol table. The symbol table includes the names of all functions and global variables, the names of all the functions the called, and their addresses.

$ *objdump -d p*

Disassemble all of the code in the program. You can also just look at individual functions. Reading the assembler code can tell you how the program works.

gdb

>*gdb p*

(gdb) *disassemble sum*

(gdb]) *x /13b sum* (examine the 13 bytes starting at sum)

strings

$ *strings –t x p*

Displays the printable strings in your program.

Object Code

```
0x401040 <sum>:
0x55
0x89
0xe5
0x8b
0x45
0x0c
0x03
0x45
0x08
0x89
0xec
0x5d
0xc3
```

Disassembled version

```
00401040 <_sum>:
 0:    55          push %ebp
 1:    89 e5       mov %esp,%ebp
 3:    8b 45 0c    mov 0xc(%ebp),%eax
 6:    03 45 08    add 0x8(%ebp),%eax
 9:    89 ec       mov %ebp,%esp
 b:    5d pop      %ebp
 c:    c3          ret
```