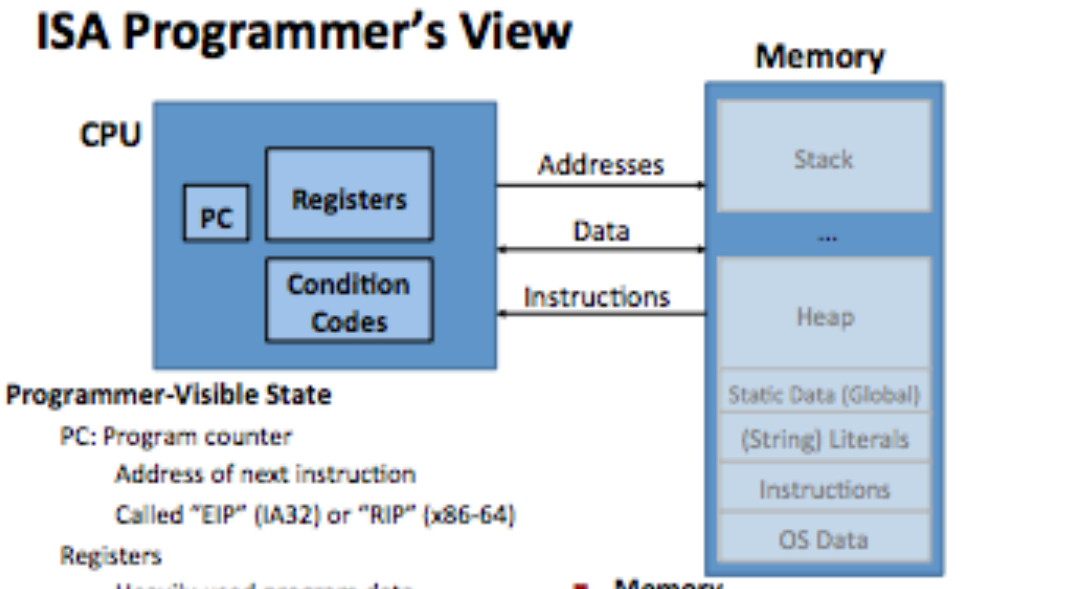
CS240 Laboratory 7

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



Programmer-Visible State

PC: Program counter

Address of next instruction

Called "EIP" (IA32) or "RIP" (x86-64)

Registers

Heavily used program data

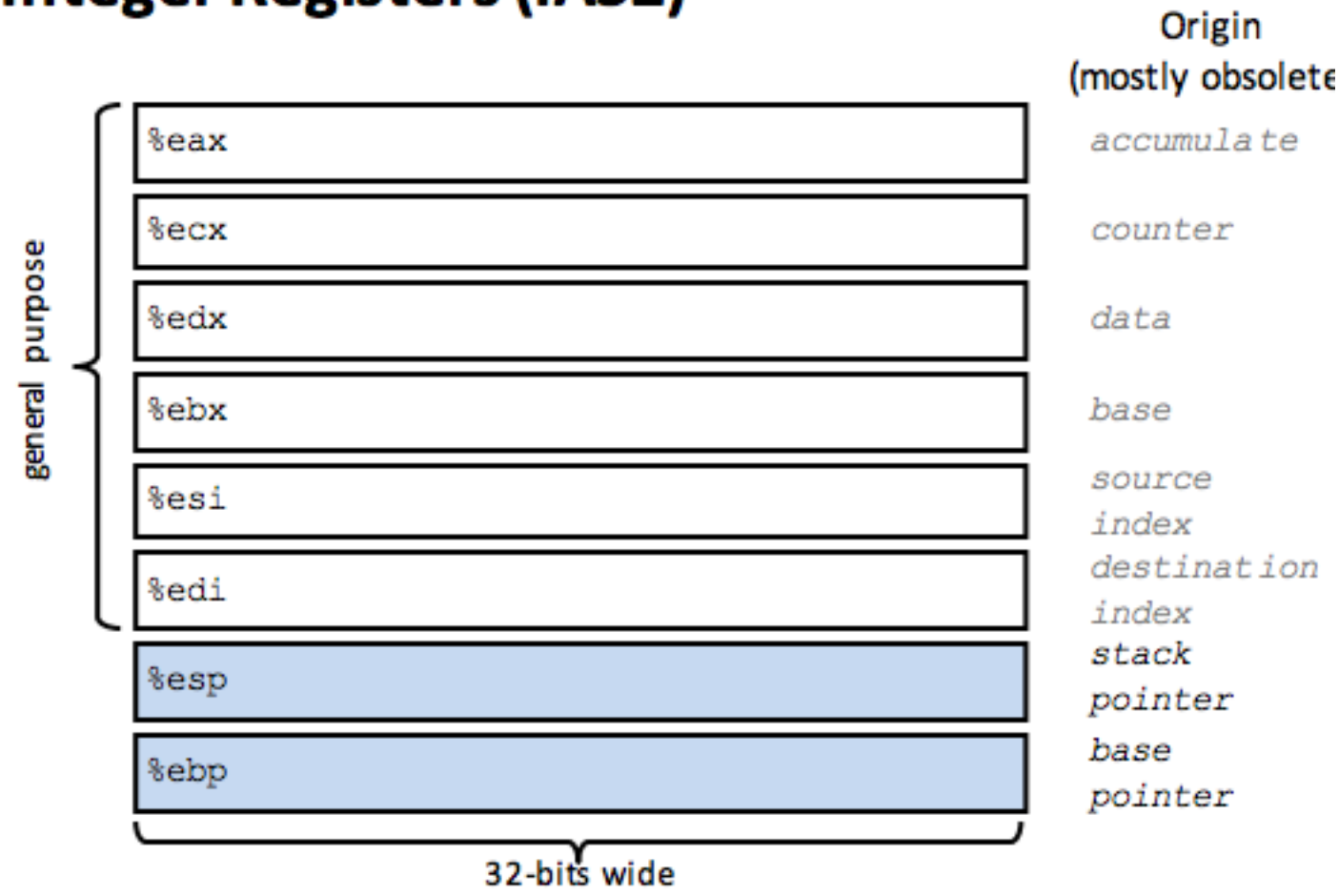
Condition codes

Store status information about most recent arithmetic operation

Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures (we'll come back to that)

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)*

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

C to X86

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

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx

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

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

Set Up

Body

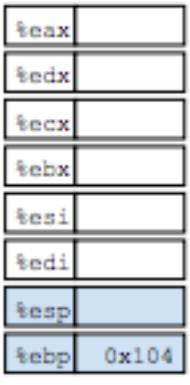
Finish

- 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!

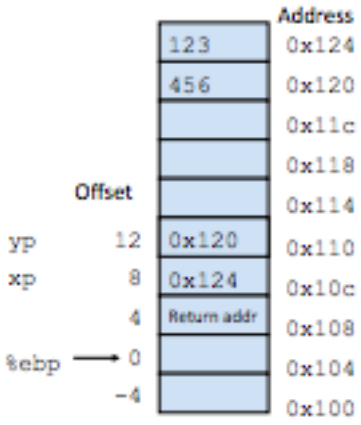
Register to Variable mapping

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

Register Values



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
- Starting 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
```