HARVARD
School of Engineering and Applied Sciences

Machine Programming 1: Introduction

CS61, Lecture 3

Prof. Stephen Chong

September 8, 2011

Announcements (1/2)

- Assignment 1 due Tuesday
  - Please fill in survey by **5pm today**!
- Assignment 2 will be released tonight
  - More information on website this evening
- Office hours will start next week
  - See course website for office hours
    - Both Announcements, and Course Staff pages.
  - More course staff coming on board soon, more office hours
- Name tags
  - At back of room
  - Fill in, put in front of you, leave at end of class *in appropriate box*

Announcements (2/2)

- Sections will start next week
  - Times:
    - Mondays 10:00-11:30am
    - Mondays 2:30-4:00pm
    - Mondays 4:00-5:30pm
    - Tuesdays 10:00-11:30am
    - Tuesdays 7:00-8:30pm (Quad)
  - College students: please complete sectioning tool by **Friday 5pm**
    - Go to `https://www.section.fas.harvard.edu/`
  - (Extension School students: Rob will be in touch to schedule your section.)
  - More info about sections on course website.

Topics for today

- C, assembly, machine code
  - C to machine code
  - Disassembly
- Assembly basics
  - Operands
  - Moving data
  - Arrays
  - LEAL: Load Effective Address
  - Data operations
  - Data types
  - x86-64

Highscore

```
-bash-3.2$ ~stephenchong/highscore
Usage: /home/s/t/stephenchong/highscore i j k l
   where i,j,k,l are integers.
   Try to get as high a score as you can.
   Note: any positive score will send an email to Prof. Chong.
```

- 35 people tried it out
- 5 people hacked the score without supplying right inputs
  - Including one score of “Infinity”
- 9 got a score of 61!
  - Requires disassembly of four functions
- 9 got more (waaaaaay more) than 61…
  - Requires understanding integer overflow!

Turning C into machine code

C program (myprog.c)

```c
int dosum(int i, int j) {
    return i+j;
}
```

↓

C compiler (gcc)

↓

Assembly program (myprog.s)

```
dosum:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        popl    %ebp
         ret
```

↓

Assembler (gas)

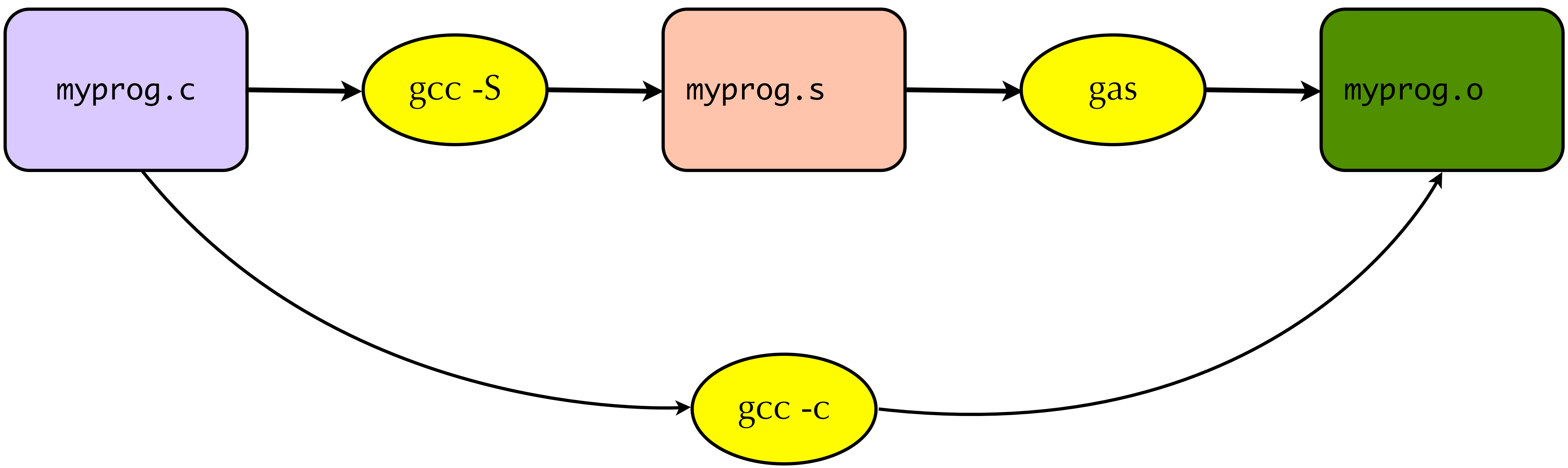
↓

Machine code (myprog.o)

```
80483b0: 55 89 e5 8b 45 0c 03 45 08 5d c3
```

Skipping assembly language

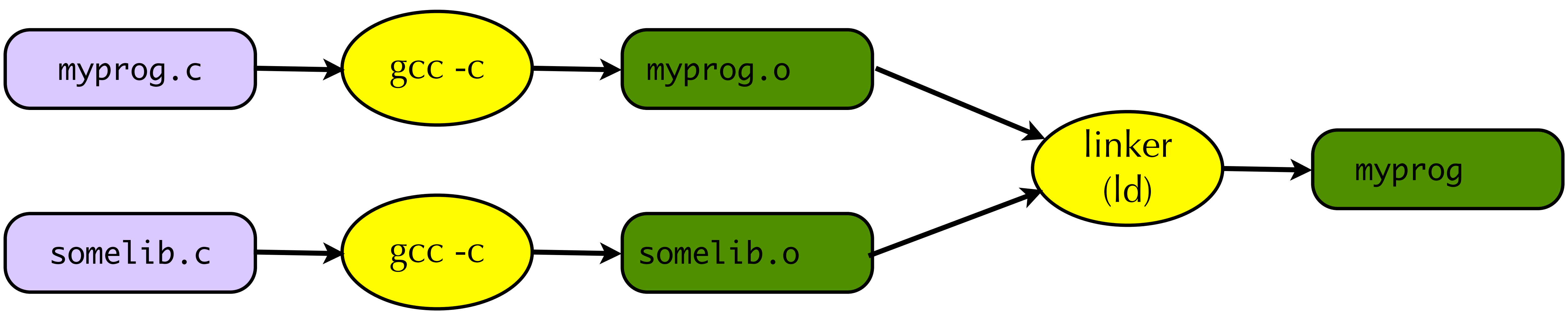
- Most C compilers generate machine code (object files) directly.
  - That is, without actually generating the human-readable assembly file.
  - Assembly language is mostly useful to people, not machines.



- Can generate assembly from C using "`gcc -S`"
  - And then compile to an object file by hand using "`gas`"

Object files and executables

- C source file (myprog.c) is compiled into an **object file** (myprog.o)
    - Object file contains the machine code for that C file.
    - It may contain references to external variables and routines
    - E.g., if myprog.c calls printf(), then myprog.o will contain a reference to printf().
- Multiple object files are **linked** to produce an executable file.
    - Typically, standard libraries (e.g., "libc") are included in the linking process.
    - Libraries are just collections of pre-compiled object files, nothing more!



Characteristics of assembly language

- Assembly language is very, very simple.
- Simple, minimal data types
  - Integer data of 1, 2, 4, or 8 bytes
  - Floating point data of 4, 8, or 10 bytes
  - No aggregate types such as arrays or structures!
- Primitive operations
  - Perform arithmetic operation on registers or memory (add, subtract, etc.)
  - Read data from memory into a register
  - Store data from register into memory
  - Transfer control of program (jump to new address)
  - Test a control flag, conditional jump (e.g., jump only if zero flag set)
- More complex operations must be built up as (possibly long) sequences of instructions.

Why you need to understand assembly language

- These days, very few people write assembly code
  - Very *very* few people write significant amounts of assembly code!
  - You won't need to write assembly in this course, and probably won't in future
- But, you will need to be able to read it to understand what a program is really doing, and how the processor works.
- Examples:
  - Understanding strange memory bugs (stack smashing, core dumps, etc.)
  - Understanding what affects the performance of a given piece of code
  - Understanding what the heck the compiler is doing to your precious C program
- Other uses...
  - Writing device drivers: Sometimes need to drop down to assembler
  - Writing an OS or embedded system
  - Writing a compiler

Disassembling

- Assembly is a human readable form of machine code
  - **Assemblers** (e.g., `gas`) compile assembly to machine code
- **Disassemblers** convert machine code to assembly
  - Interprets bits as instructions
  - Useful tools for examining machine code

Disassemblers

- `objdump`
  - `objdump -d myprog.o`
  - Can be used on object files (.o) or complete executables
- `gdb`
  - GNU debugger
  - Can disassemble, run, set breakpoints, examine memory and registers
  - Course website contains links to some resources for learning gdb
- Play around with both! `gdb` will be especially helpful in assignments

What can be disassembled?

- Anything that can be interpreted as executable code
- Disassembler simply examines bits, interprets them as machine code, and reconstructs assembly

```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:     55                  push   %ebp
30001001:     8b ec               mov    %esp,%ebp
30001003:     6a ff               push   $0xffffffff
30001005:     68 90 10 00 30      push   $0x30001090
3000100a:     68 91 dc 4c 30      push   $0x304cdc91
```

Topics for today

- C, assembly, machine code
  - C to machine code
  - Disassembly
- Assembly basics
  - Operands
  - Moving data
  - Arrays
  - LEAL: Load Effective Address
  - Data operations
  - Data types
  - x86-64

Addressing modes

- Most instructions have one or more **operands**
  - Specify input and output for operations
  - Inputs can be registers, memory locations, or immediate (constant) values
  - Outputs can be saved to registers or memory locations
- Collectively, these ways of accessing operands are called **addressing modes**
- Different instructions support different addressing modes
  - Need to check the manual to find out which modes are allowed
  - Example: "`movl`" instruction (copy 32-bit value) supports...
    - Immediate to register `movl $0x1000, %eax`
    - Register to register `movl %eax, %ebx`
    - Memory to register (a.k.a. "load") `movl (%eax), %ebx`
    - Register to memory (a.k.a. "store") `movl %eax, (%ebx)`
    - Cannot move from memory to memory!

Immediate and register operands

- Immediate operands are for constant values
  - Written with a `$` followed by integer in standard C notation
  - E.g., `$-577`, `$0x1F`
  - Operand value is simply the immediate value
- Register operands denote content of register
  - Written as the name of the register, which starts with a `%` sign
  - E.g., `%eax`, `%ebx`
  - Operand value is $R[E_a]$ where E_a denotes a register, $R[E_a]$ denotes value stored in register

Memory operands

- Most general form is $Imm(E_b, E_i, s)$
  - Imm is immediate offset, E_b is base register, E_i is index register, s is scale (must be 1, 2, 4 or 8)
  - Effective address is $Imm + R[E_b] + R[E_i]\times s$
  - Operand value is $M[\ Imm + R[E_b] + R[E_i]\times s\ ]$
- Other forms special cases of this general form
  - Imm is an immediate, or **absolute**, address
    - e.g., `0x1a38`
  - (E_b) is an indirect address
    - e.g., `(%eax)` is contents of register `%eax`
  - $Imm(E_b)$ is a base address plus a displacement
    - e.g., `0x8(%ebp)` is contents of register `%ebp` plus 8
  - (E_b, E_i) and $Imm(E_b, E_i)$ are indexed addresses

Address computation example

`%edx`	`0xf000`
`%ecx`	`0x100`

Expression	Computation	Address
`0x8(%edx)`		
`(%edx, %ecx)`		
`(%edx, %ecx, 4)`		
`0x80(, %edx, 2)`		

Address computation example

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx, %ecx)		
(%edx, %ecx, 4)		
0x80(, %edx, 2)		

Address computation example

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx, %ecx)	0xf000 + 0x100	0xf100
(%edx, %ecx, 4)		
0x80(, %edx, 2)		

Address computation example

%edx	0xf000
%ecx	0x100

Expression	**Computation**	**Address**
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx, %ecx)	0xf000 + 0x100	0xf100
(%edx, %ecx, 4)	0xf000 + 4*0x100	0xf400
0x80(, %edx, 2)		

Address computation example

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx, %ecx)	0xf000 + 0x100	0xf100
(%edx, %ecx, 4)	0xf000 + 4*0x100	0xf400
0x80(, %edx, 2)	2*0xf000 + 0x80	0x1e080

Moving data

- Copy data from one location to another
  - Heavily used!
- `mov`*x* *source*, *dest*
  - *x* is one of `b`, `w`, `l`
  - `movb` *source*, *dest*
    - Move 1-byte "byte"
  - `movw` *source*, *dest*
    - Move 2-byte "word" (for historical reasons)
  - `movl` *source*, *dest*
    - Move 4-byte "long word" (for historical reasons)

AT&T vs Intel syntax

- Two common ways of formatting IA32 assembly
  - AT&T
    - We use this in class, used by gcc, gdb, objdump
  - Intel
    - Used by Intel documentation, Microsoft tools
- Differences:
  - Intel omits size designation: `mov` instead of `movl`
  - Intel omits % from register names: `ebp` instead of `%ebp`
  - Intel describes memory locations differently: `[ebp+8]` instead of `8(%ebp)`
  - **Intel lists operands in reverse order:** `mov` *dest*`,` *src* instead of `movl` *src*`,` *dest*

`movl` examples

Instruction	Src	Dest	C analog
`movl $0x4,%eax`			
`movl $-147,(%eax)`			
`movl %eax,%edx`			
`movl %eax,(%edx)`			
`movl (%eax),%edx`			

`movl` examples

Instruction	Src	Dest	C analog
`movl $0x4,%eax`	Imm	Reg	`temp = 0x4;`
`movl $-147,(%eax)`			
`movl %eax,%edx`			
`movl %eax,(%edx)`			
`movl (%eax),%edx`			

`movl` examples

Instruction	Src	Dest	C analog
`movl $0x4,%eax`	Imm	Reg	`temp = 0x4;`
`movl $-147,(%eax)`	Imm	Mem	`*p = -147;`
`movl %eax,%edx`			
`movl %eax,(%edx)`			
`movl (%eax),%edx`			

`movl` examples

Instruction	Src	Dest	C analog
`movl $0x4,%eax`	Imm	Reg	`temp = 0x4;`
`movl $-147,(%eax)`	Imm	Mem	`*p = -147;`
`movl %eax,%edx`	Reg	Reg	`temp2 = temp1;`
`movl %eax,(%edx)`			
`movl (%eax),%edx`			

`movl` examples

Instruction	Src	Dest	C analog
`movl $0x4,%eax`	Imm	Reg	`temp = 0x4;`
`movl $-147,(%eax)`	Imm	Mem	`*p = -147;`
`movl %eax,%edx`	Reg	Reg	`temp2 = temp1;`
`movl %eax,(%edx)`	Reg	Mem	`*p = temp;`
`movl (%eax),%edx`			

`movl` examples

Instruction	Src	Dest	C analog
`movl $0x4,%eax`	Imm	Reg	`temp = 0x4;`
`movl $-147,(%eax)`	Imm	Mem	`*p = -147;`
`movl %eax,%edx`	Reg	Reg	`temp2 = temp1;`
`movl %eax,(%edx)`	Reg	Mem	`*p = temp;`
`movl (%eax),%edx`	Mem	Reg	`temp = *p;`

Note: Cannot move directly from memory to memory with single instruction!

Note: C pointers are just memory addresses

Example: swap

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

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

  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

Example: swap

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

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

  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

Understanding swap

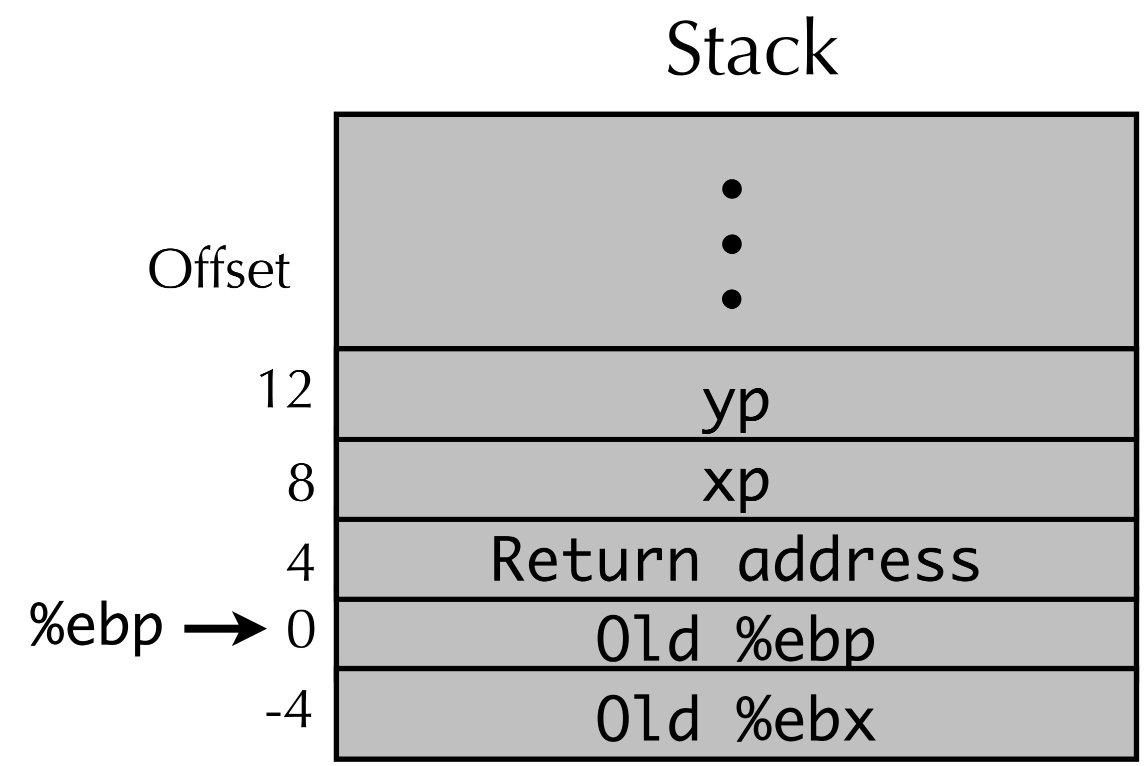
```
void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t0;
  *yp = t1;
}
```

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

Body

Understanding swap

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

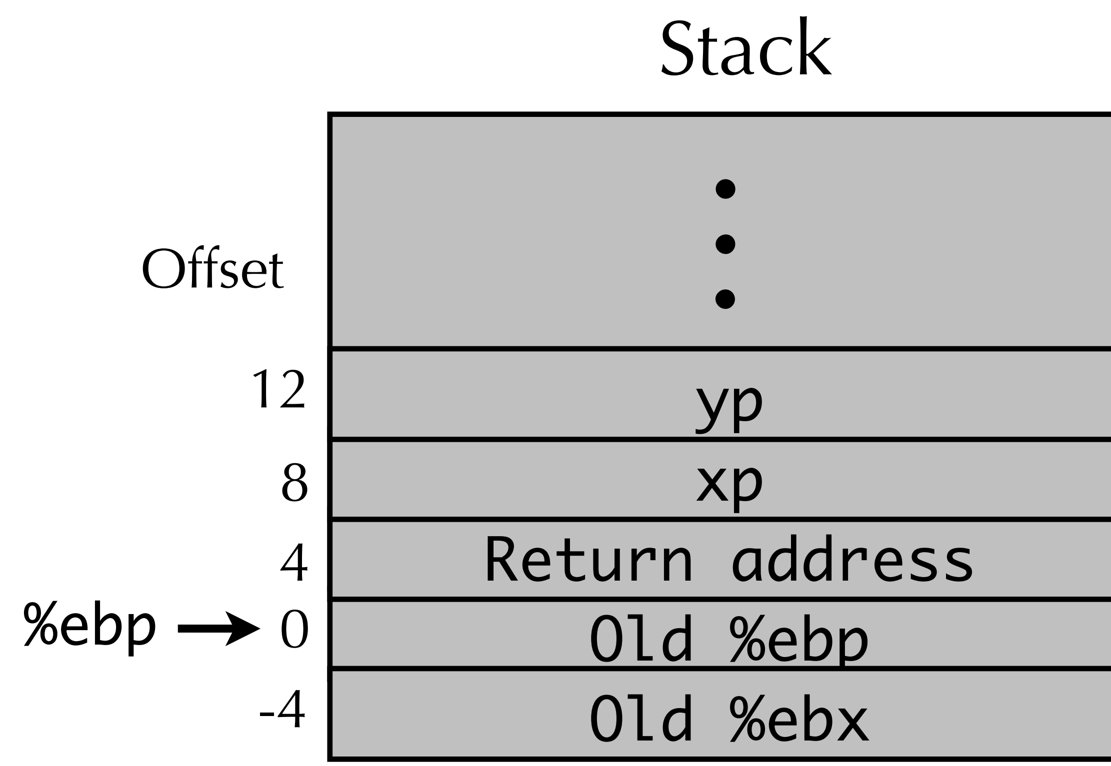



```
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
```

Body

Understanding swap

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



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

```
movl 12(%ebp),%ecx   # %ecx = yp
movl 8(%ebp),%edx    # %edx = xp
movl (%ecx),%eax     # %eax = *yp
movl (%edx),%ebx     # %ebx = *xp
movl %eax,(%edx)     # *xp = %eax
movl %ebx,(%ecx)     # *yp = %ebx
```

Body

Understanding swap

Register	Value
%eax	
%ecx	0x11c
%edx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Stack

	Offset	Stack	Address
		123	0x120
		456	0x11c
			0x118
			0x114
	12	0x11c	0x110
	8	0x120	0x10c
	4	Return address	0x108
%ebp →	0	Old %ebp	0x104
	-4	Old %ebx	0x100

Body

```
movl 12(%ebp),%ecx    # %ecx = yp
movl 8(%ebp),%edx     # %edx = xp
movl (%ecx),%eax      # %eax = *yp
movl (%edx),%ebx      # %ebx = *xp
movl %eax,(%edx)      # *xp = %eax
movl %ebx,(%ecx)      # *yp = %ebx
```

Understanding swap

Register	Value
%eax	
%ecx	0x11c
%edx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Offset	Stack	Address
	123	0x120
	456	0x11c
		0x118
		0x114
12	0x11c	0x110
8	0x120	0x10c
4	Return address	0x108
%ebp → 0	Old %ebp	0x104
-4	Old %ebx	0x100

```
movl 12(%ebp),%ecx   # %ecx = yp
movl 8(%ebp),%edx    # %edx = xp
movl (%ecx),%eax     # %eax = *yp
movl (%edx),%ebx     # %ebx = *xp
movl %eax,(%edx)     # *xp = %eax
movl %ebx,(%ecx)     # *yp = %ebx
```

Body

Understanding swap

Register	Value
%eax	456
%ecx	0x11c
%edx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Stack

Offset	Stack	Address
	123	0x120
	456	0x11c
		0x118
		0x114
12	0x11c	0x110
8	0x120	0x10c
4	Return address	0x108
%ebp → 0	Old %ebp	0x104
-4	Old %ebx	0x100

Body

```
movl 12(%ebp),%ecx   # %ecx = yp
movl 8(%ebp),%edx    # %edx = xp
movl (%ecx),%eax     # %eax = *yp
movl (%edx),%ebx     # %ebx = *xp
movl %eax,(%edx)     # *xp = %eax
movl %ebx,(%ecx)     # *yp = %ebx
```

Understanding swap

Register	Value
%eax	456
%ecx	0x11c
%edx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

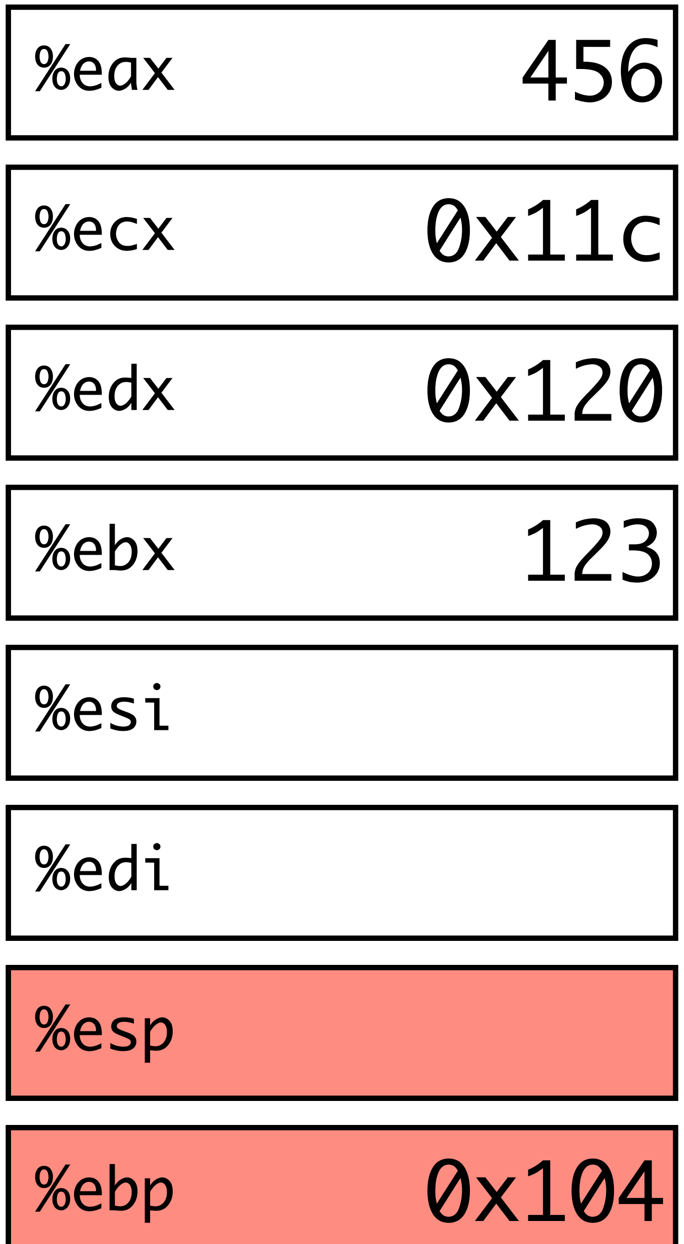
Stack

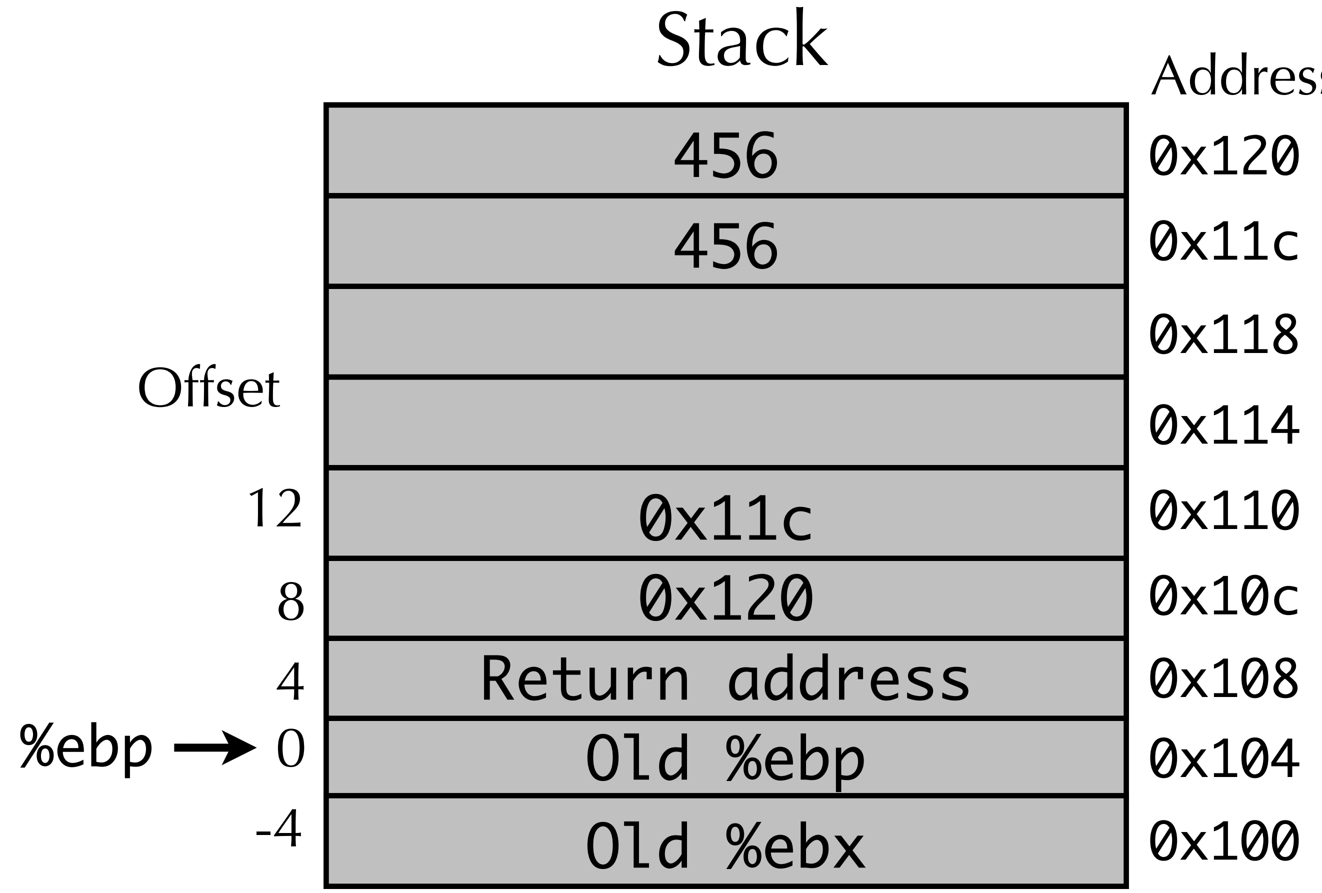
Offset	Stack	Address
	123	0x120
	456	0x11c
		0x118
		0x114
12	0x11c	0x110
8	0x120	0x10c
4	Return address	0x108
%ebp → 0	Old %ebp	0x104
-4	Old %ebx	0x100

```
movl 12(%ebp),%ecx   # %ecx = yp
movl 8(%ebp),%edx    # %edx = xp
movl (%ecx),%eax     # %eax = *yp
movl (%edx),%ebx     # %ebx = *xp
movl %eax,(%edx)     # *xp = %eax
movl %ebx,(%ecx)     # *yp = %ebx
```

Body

Understanding swap






```
movl 12(%ebp),%ecx   # %ecx = yp
movl 8(%ebp),%edx    # %edx = xp
movl (%ecx),%eax     # %eax = *yp
movl (%edx),%ebx     # %ebx = *xp
movl %eax,(%edx)     # *xp = %eax
movl %ebx,(%ecx)     # *yp = %ebx
```

Body

Understanding swap

Register	Value
%eax	456
%ecx	0x11c
%edx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

Stack

Offset	Stack	Address
	456	0x120
	123	0x11c
		0x118
		0x114
12	0x11c	0x110
8	0x120	0x10c
4	Return address	0x108
%ebp → 0	Old %ebp	0x104
-4	Old %ebx	0x100

Body

```
movl 12(%ebp),%ecx   # %ecx = yp
movl 8(%ebp),%edx    # %edx = xp
movl (%ecx),%eax     # %eax = *yp
movl (%edx),%ebx     # %ebx = *xp
movl %eax,(%edx)     # *xp = %eax
movl %ebx,(%ecx)     # *yp = %ebx
```

Topics for today

- C, assembly, machine code
  - C to machine code
  - Disassembly
- Assembly basics
  - Operands
  - Moving data
  - Arrays
  - LEAL: Load Effective Address
  - Data operations
  - Data types
  - x86-64

Arrays

- `int a[10]` declares an array of 10 integers in C
  - `a[0]` is the first element of the array, `a[9]` is the 10th.
- But can use other indices than 0..9!
  - E.g.,

```
void foo(int a[10]) {
  a[-1] = a[20] = 0x42; // Not good!!
}
```

- Also, arrays and pointers are very closely related
  - E.g.,

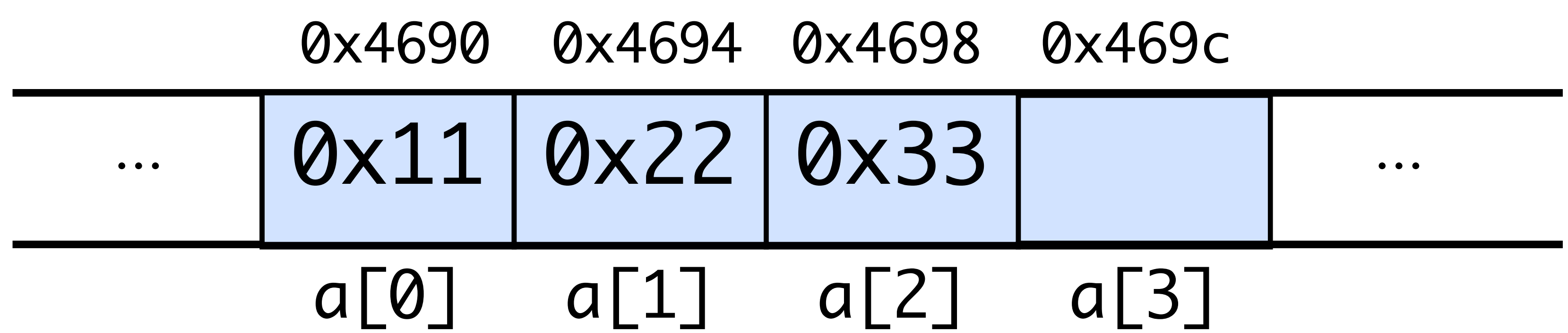
```
void bar(int a[10]) {
  int *p = a;
  int i;
  for (i = 0; i < 10; i++)
      *(p++) = 0;
}
```

- What's going on?

Arrays

- An array is a contiguous region of memory

```
int a[4];
a[0] = 0x11;
a[1] = 0x22;
a[2] = 0x33;
```



- Can be thought of as a pointer to the first element of the array

```
movl $0x4690, %eax   ; Set %eax to address of 'a'
movl $0x11, (%eax)   ; a[0] = 0x11
movl $0x22, 4(%eax) ; a[1] = 0x22
movl $0x33, 8(%eax) ; a[2] = 0x33
```

- Why do we add 4 to `%eax` each time?
  - Each integer is represented with 4 bytes, so the address of `a[1]` and `a[2]` differ by 4

Arrays

```
void foo(int a[10]) {
   a[-1] = 0x42;    // Not good!!
   a[20] = 0xbeef; // Not good!!
}
```

```
movl   0x8(%ebp),%eax   ; 0x8(%ebp) contains the pointer to array ‘a’
subl   $0x4,%eax        ; a[-1]
movl   $0x42,(%eax)     ; a[-1] = 0x42
movl   0x8(%ebp),%eax
addl   $0x50,%eax       ; a[20]
movl   $0xbeef,(%eax)   ; a[20] = 0xbeef
```

```
void bar(int a[10]) {
   int *p = a;
   *(p+3) = 0xcafe;
}
```

```
movl   0x8(%ebp),%eax   ; 0x8(%ebp) contains the pointer to array ‘a’
addl   $0xc,%eax        ; p+3
movl   $0xcafe,(%eax)   ; *(p+3) = 0xcafe. Same as a[3] = 0xcafe
```

Address computation instruction

- `leal` *src*`,` *dest*
  - *src* is address mode expression
    - e.g., `(%eax)` or `0x8(%ebp)`
    - Most generally, *Imm*(*base*, *index*, *scale*)
  - Set *dest* to the address denoted by expression *src*
  - "Load effective address"
- Can compute an address without a memory reference
  - E.g., compilation of C code `p = &x[i]`

```
leal (%eax, %ebx, 4), %edx
```

- Can also be used to compute arithmetic expressions!
  - Anything of the form $x + y*k$, where $k = 1, 2, 4$, or 8

Some arithmetic operations

- Two operand instructions

Format		Equivalent C computation	
`addl`	*Src,Dest*	*Dest = Dest + Src*	
`subl`	*Src,Dest*	*Dest = Dest - Src*	
`imull`	*Src,Dest*	*Dest = Dest * Src*	
`sall`	*Src,Dest*	*Dest = Dest << Src*	Also called `shll`
`sarl`	*Src,Dest*	*Dest = Dest >> Src*	Arithmetic
`shrl`	*Src,Dest*	*Dest = Dest >> Src*	Logical
`xorl`	*Src,Dest*	*Dest = Dest ^ Src*	
`andl`	*Src,Dest*	*Dest = Dest & Src*	
`orl`	*Src,Dest*	*Dest = Dest | Src*	

- No distinction between signed and unsigned int. Why?

Shifting

- There is only one left-shift operator
  - `sal` and `shl` are synonyms
  - Shift bits left, filling from the right with zeros
  - E.g., `0x1 << 3` $= 1 \times 2^3 =$ `0x8`
- Two different right-shift operators: `sar` and `shr`
  - Both correspond to C operator `>>`
  - What's the difference?

Shifting

- `shr` (logical right-shift) fills from left with **zeros**
- `sar` (arithmetic right-shift) files from left with **sign bit** of operand
  - A form of **sign extension**
- C compiler figures out which to use based on type of operand
  - Unsigned int uses logical right shift; signed int uses arithmetic right shift

```
#define print_unsigned(i) printf("%u (%x)\n",(i),(i))
#define print_signed(i) printf("%d (%x)\n",(i),(i))
int main() {
  int s = 0x80000000;
  unsigned int u = 0x80000000;
  print_unsigned(u);
  print_unsigned(u >> 1); // Uses shrl
  print_signed(s);
  print_signed(s >> 1);   // Uses sarl
  return 1;
}
```

```
2147483648 (80000000)
1073741824 (40000000)
-2147483648 (80000000)
-1073741824 (c0000000)
```

Some more arithmetic operations

- One operand instructions

Format	**Equivalent C computation**
`incl` *Dest*	*Dest* = *Dest* + 1
`decl` *Dest*	*Dest* = *Dest* - 1
`negl` *Dest*	*Dest* = -*Dest*
`notl` *Dest*	*Dest* = ~*Dest*

- See textbook for more instructions

Example: logical

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set up

Body

Finish

Example: logical

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set up

Body

Finish

```
movl 8(%ebp),%eax      #   %eax = x
xorl 12(%ebp),%eax     #   %eax = x^y (t1)
sarl $17,%eax          #   %eax = t1>>17 (t2)
andl $8185,%eax        #   %eax = t2 & 8185
```

$1<<13 = 2^{13} = 8192$

$8192 - 7 = 8185$