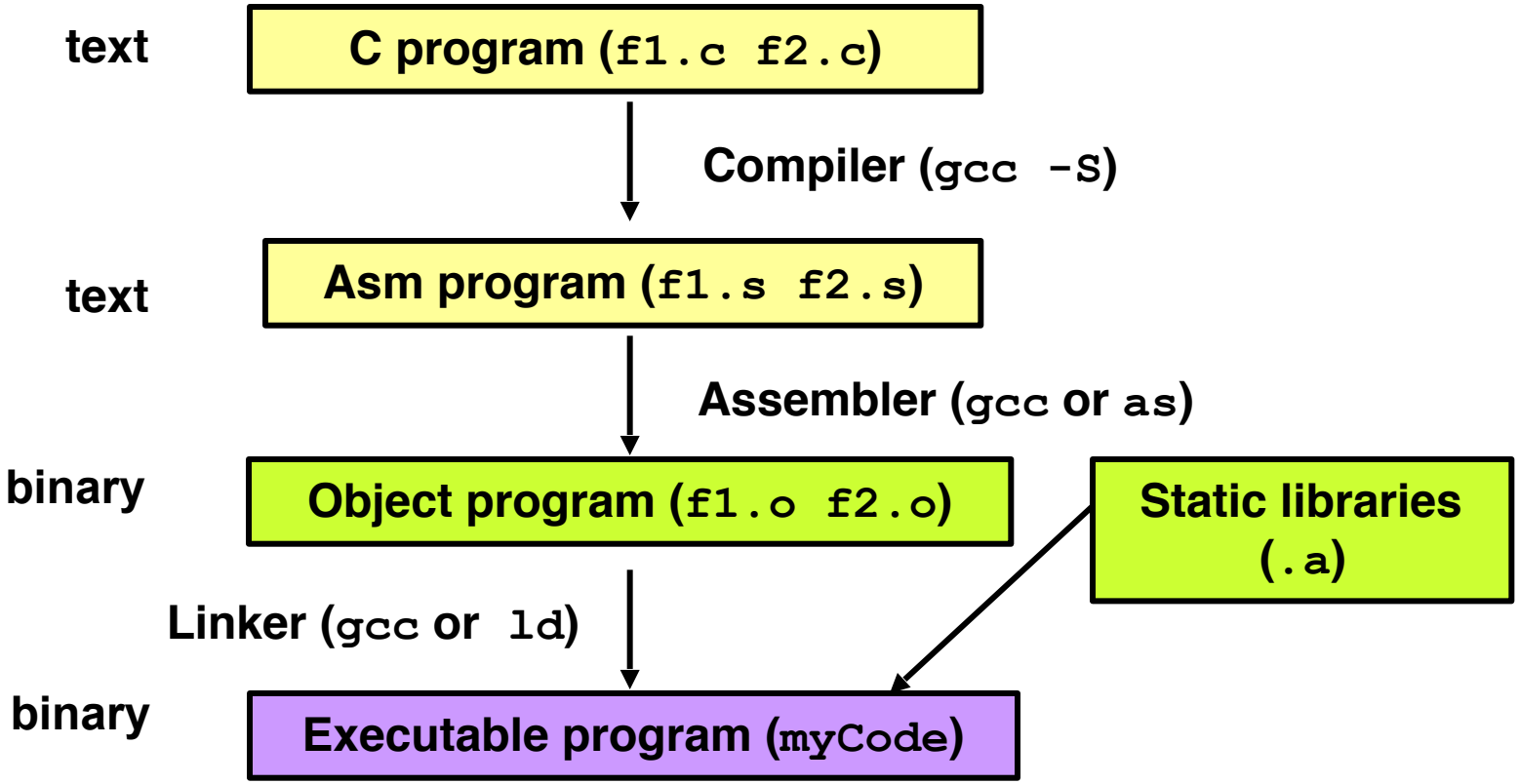
Code in files: `f1.c f2.c`

Compile with command: `gcc f1.c f2.c -o myCode`



x86 Assembly

- Three basic types of operands:
- Constants ; CPU registers ; Refs. to memory locations.
- Available data types:

Integer data of: 1 (byte), 2 (word), or 4 (long) bytes -
(note non-standard use of word (C: short) and long (C: int))

These could be addresses or data values

Floating point data of: 2 (float), 4 (double), 10 (extended)

All other (aggregate) data types: Arrays, structs

Just contiguous bytes in memory!

Available Operations:

Assembly Operations

Data Movement: Memory <--> Registers, Reg. <--> Reg.

Arithmetic/Logic: On data in memory or registers

Branching: Unconditional jumps to/from procedures
& conditional branching

Moving Data

- Typical use (GAS): `mov source, dest`
- `movl` ("long") 4 bytes of data `(also movb,movw, movq)`
- Operands:

  -Immediate: These are constants like $123, $-567, $0x804
  encoded as 1, 2 or 4 bytes

  -Register: Could be one of the eight registers **except**
  %esp and %ebp - reserved for particular use.

  -Memory: Many ways of accessing data from memory -
  these are the *Addressing Modes* of the ISA

`movl` Operands

- Source: Immediate, Destination: Register

```
movl $1234, %eax
```

C: `t = 1234`

- Source: Immediate, Destination: Memory

```
movl $1234, (%eax)
```

- Source: Register, Destination: Register

```
movl %eax, %ebx
```

- Source: Register, Destination: Memory

```
movl %eax, (%ebx)
```

- Source: Memory, Destination: Register

```
movl (%eax), %ebx
```

There is NO mem-mem transfer in a single operation.

Normal:

Register value **r** specifies memory address:

(**reg**) --> M(reg) *i.e.* `movl (%eax),%ecx`

Displacement:

Register value **reg** specifies start of mem. address

and the constant **d** (*displacement*) specifies offset

d(reg) --> M(reg + d) *i.e.* `movl 8(%ebp),%edx`

Indexed Addressing:

d(rb,ri,s) --> M(rb+d+s*ri) i.e. `movl 8(%edi, %ecx),%eax`

rb = base reg., ri = index reg. s = scale factor (1,2,4 or 8)

Sp. Cases: (a) *(rb,ri)* ***(b)*** *d(rb,ri)* ***(c)*** *(rb,ri,s)*

(See examples done in class)

Arithmetic Operations

- Two Operand Instructions
- OPERATON RESULT

`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*

One Operand Instructions

incl Dest	Dest = Dest + 1
decl Dest	Dest = Dest - 1
notl Dest	Dest = ~Dest
negl Dest	Dest = -Dest

Address Computation

There is a form of the `movl` instruction that can perform a Memory Addresses computation without a memory ref. Its destination MUST be a register.

Load Effective Address:

```
leal <epxr>, dest
```

Where <expr> is an addressing mode expression

eg `leal 6(%eax),%edx` sets %edx to %eax+6

`leal (%eax, %ecx,4),%edx` sets %edx to %eax+4*ecx

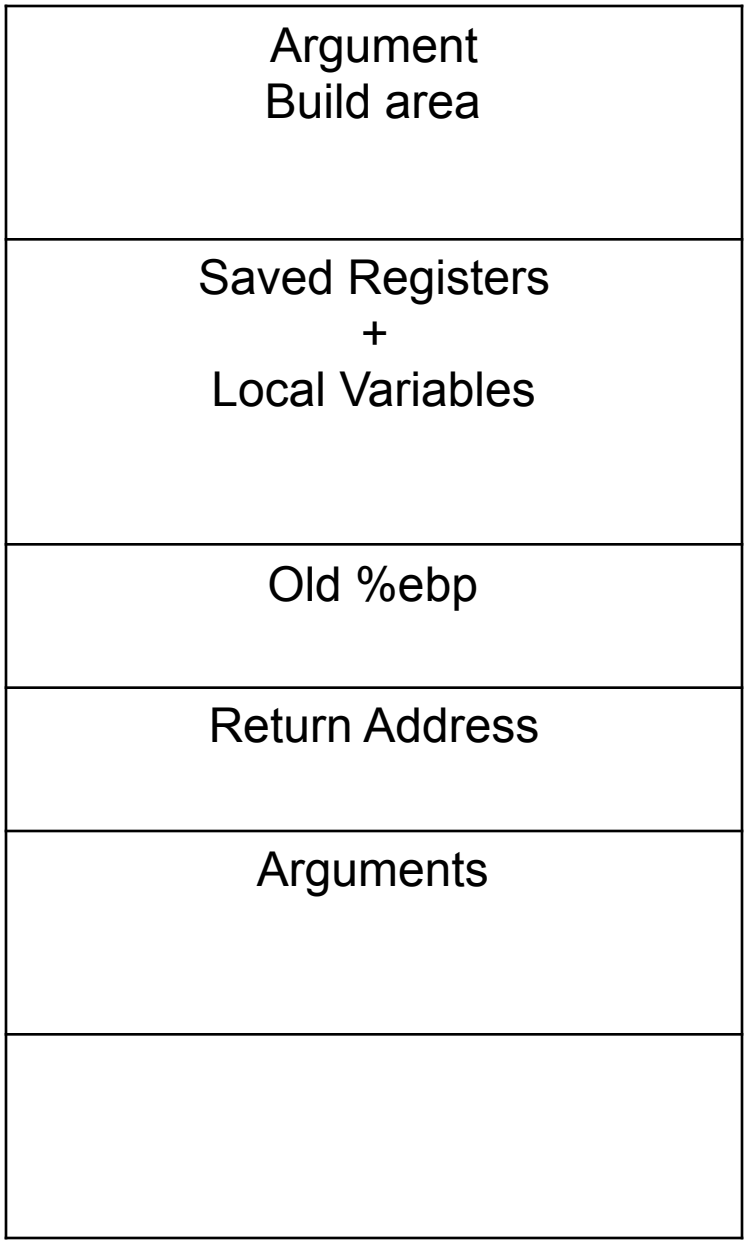
IA32 Stack

- Region of memory managed with stack discipline
- Register %esp indicates lowest allocated position in stack
- – i.e., address of top element
- Pushing
- • pushl *Src*
  - Fetch operand at *Src*
  - Decrement %esp by 4
  - Write operand at address given by %esp
- Stack “Bottom”
- Increasing Addresses
- **Stack Pointer**
-

Popping

popl Dest %esp

- Read operand at address given
- Stack Grows Down
- by %esp
- Increment %esp by 4
- Write to *Dest*


Argument
Build area
Saved Registers
+
Local Variables
Old %ebp
Return Address
Arguments

Position	Contents	Frame
4n+4(%ebp)	arg. n	
...	...	Prev
+8(%ebp)	arg. 1	
+4(%ebp)	return address	
0(%ebp)	prev. %ebp val.	Current
-4(%ebp)		
...	locals and temps	
0(%esp)		