Machine-Level Programming: Introduction

Chapter 3 of B&O

ISA - operators and stuff machine needs
Micro arch

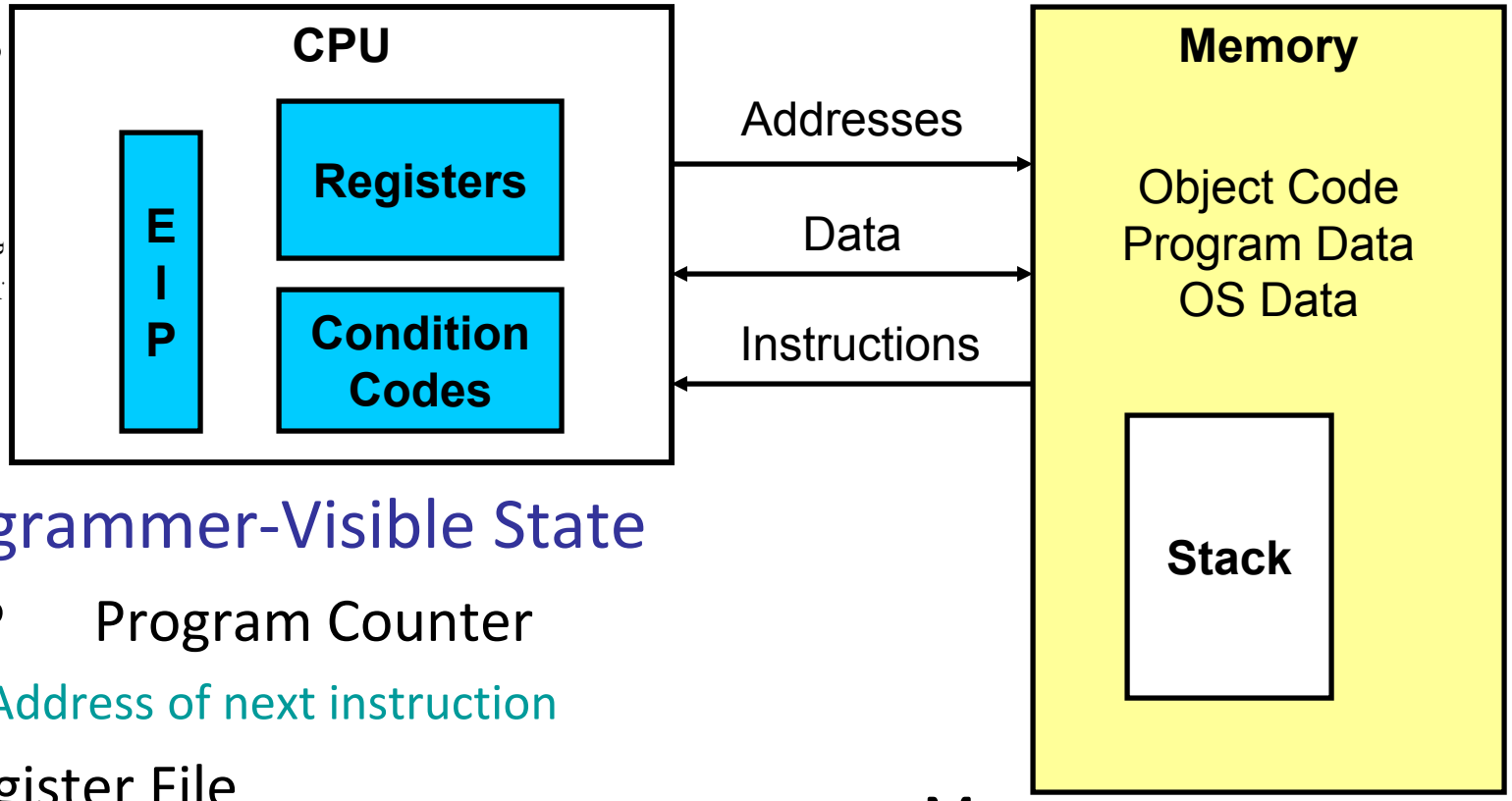
Language -> IR -> Assemll-> Machine Code -> [ISA][Micro arch]

Some notes adopted from Bryant and O'Hallaron

IA32 Processors

- Evolutionary Design
  - Starting in 1978 with 8086
  - Added more features as time goes on
  - Still support old features, although obsolete
- Complex Instruction Set Computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC) Low ops, Simple, Takes up more space (breaks up into pieces and can put in parallel)
  - But, Intel has done just that!

Assembly Programmer's View



Instructor keeping track where we are.

Registers - Compiler takes things which address and stores the same data in talking about. register so it faster if didn't

- Programmer-Visible State
  - EIP Program Counter
    - Address of next instruction
  - Register File
    - 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

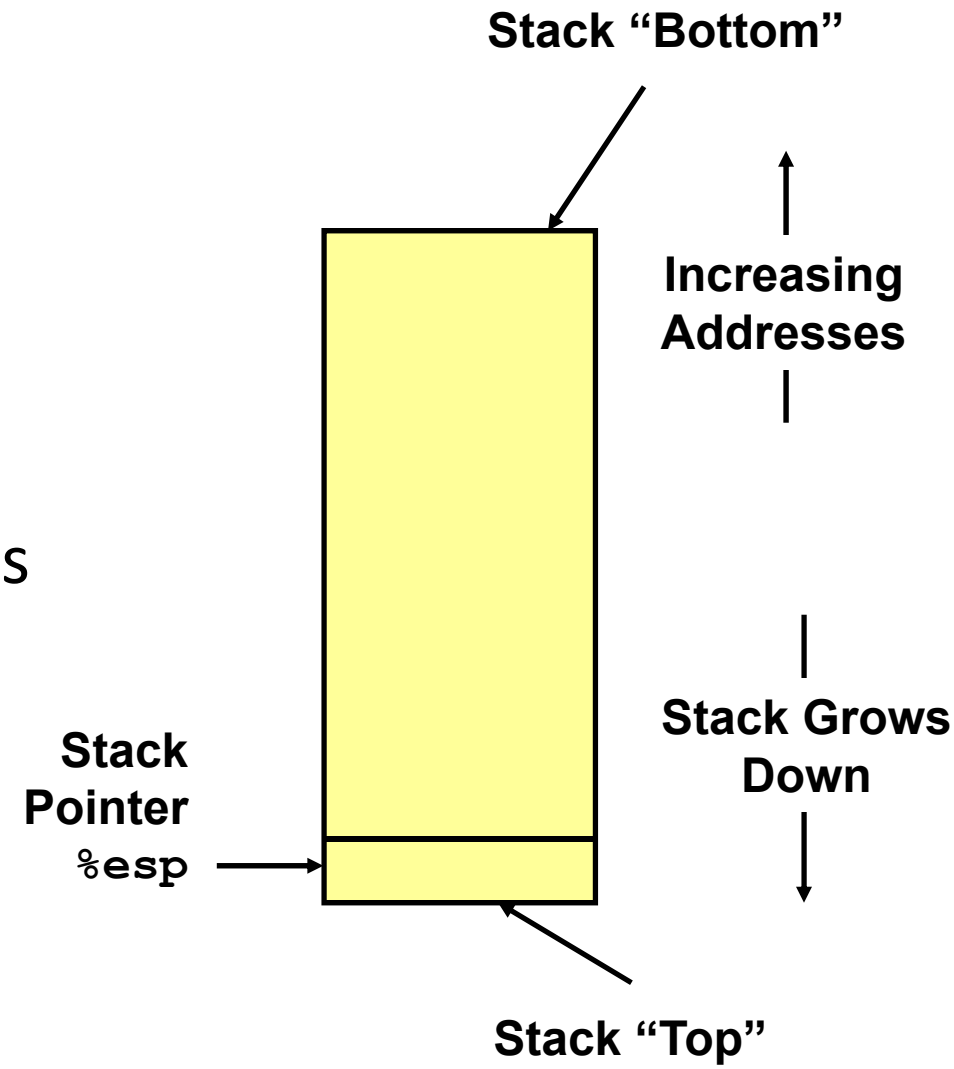
IA32 Registers

Integer registers

IA32 Stack

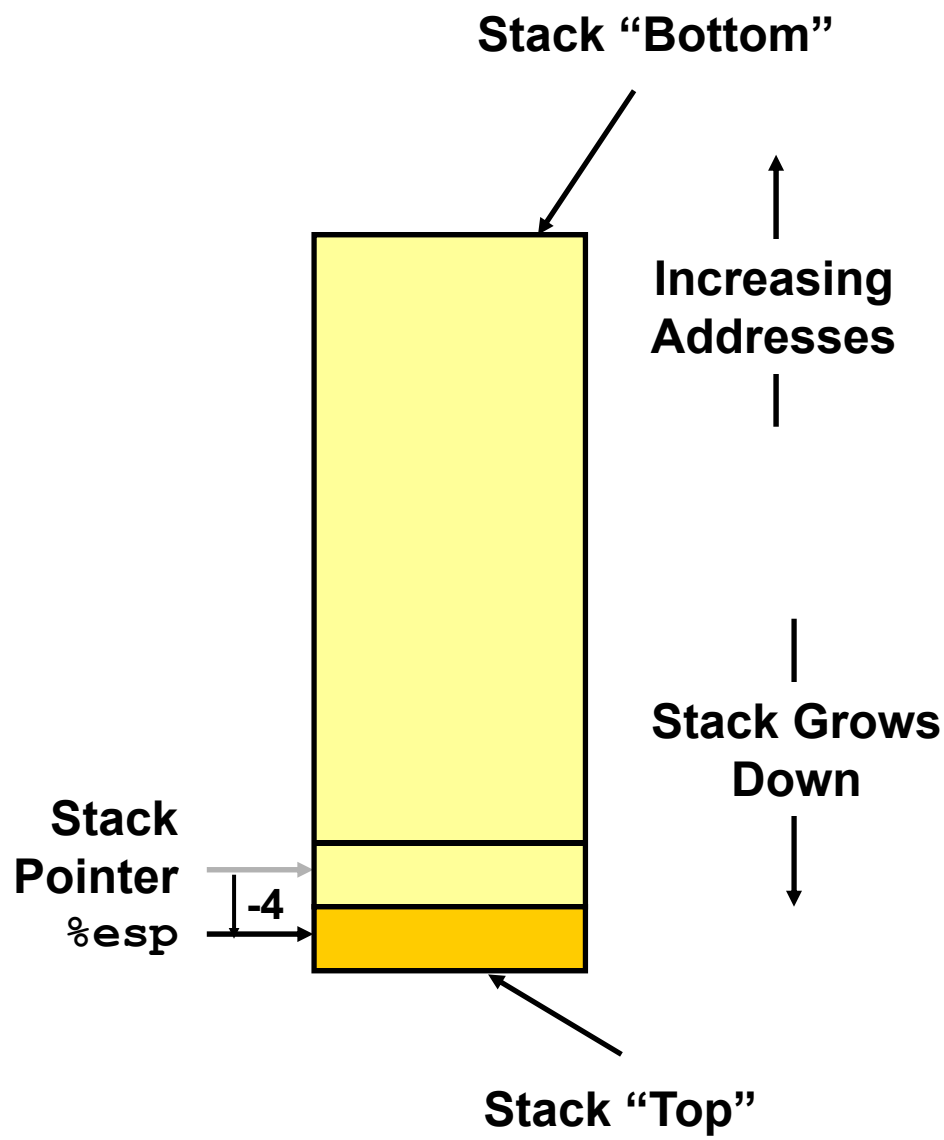
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
  - address of top element

Register is not in memory, it is in a seperate



IA32 Stack Pushing

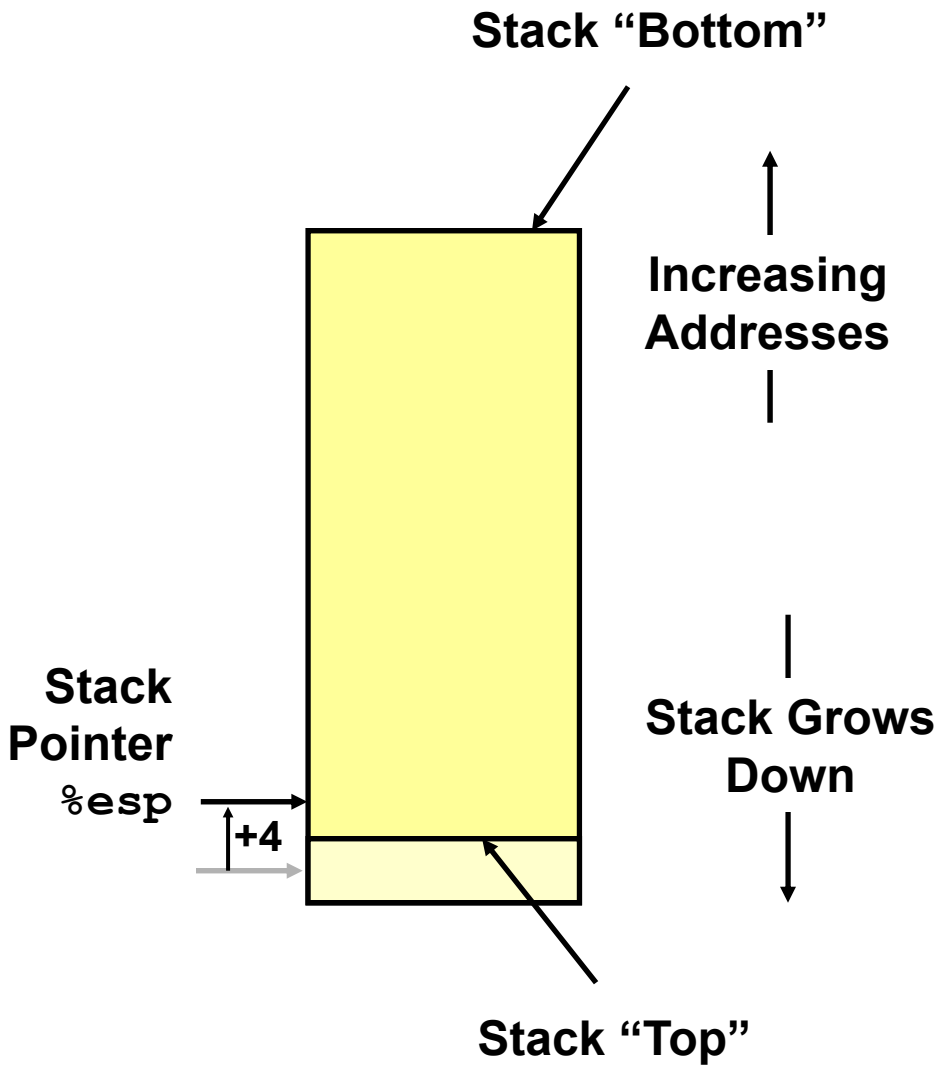
- Pushing
  - `pushl` *Src*
  - Fetch operand at *Src*
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`



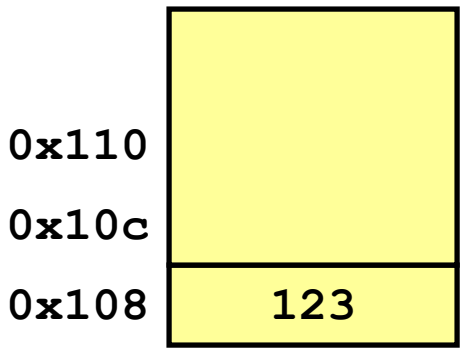
IA32 Stack Popping

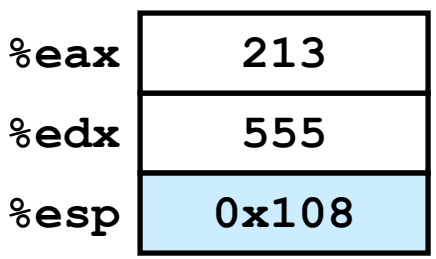
- Popping
  - `popl` *Dest*
  - Read operand at address given by `%esp`
  - Increment `%esp` by 4
  - Write to *Dest*

If you store 512 in the memory, and you pop, you don't erase that 512. You just change the esp pointin

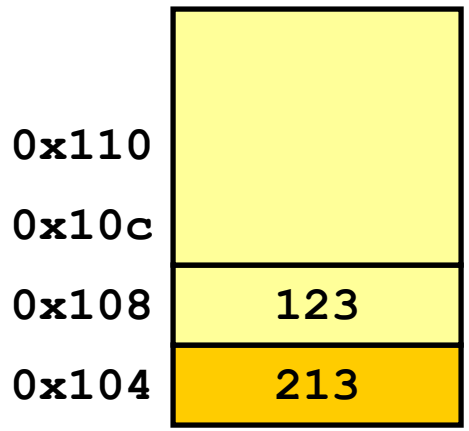


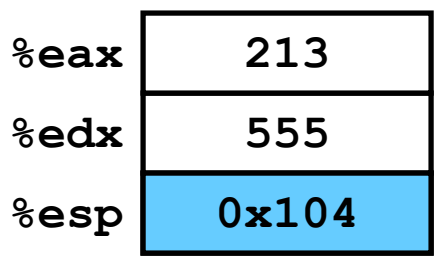
Stack Operation Examples



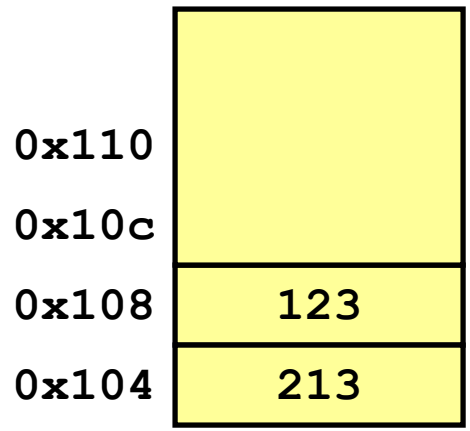


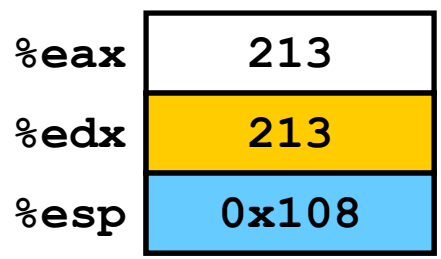
pushl %eax





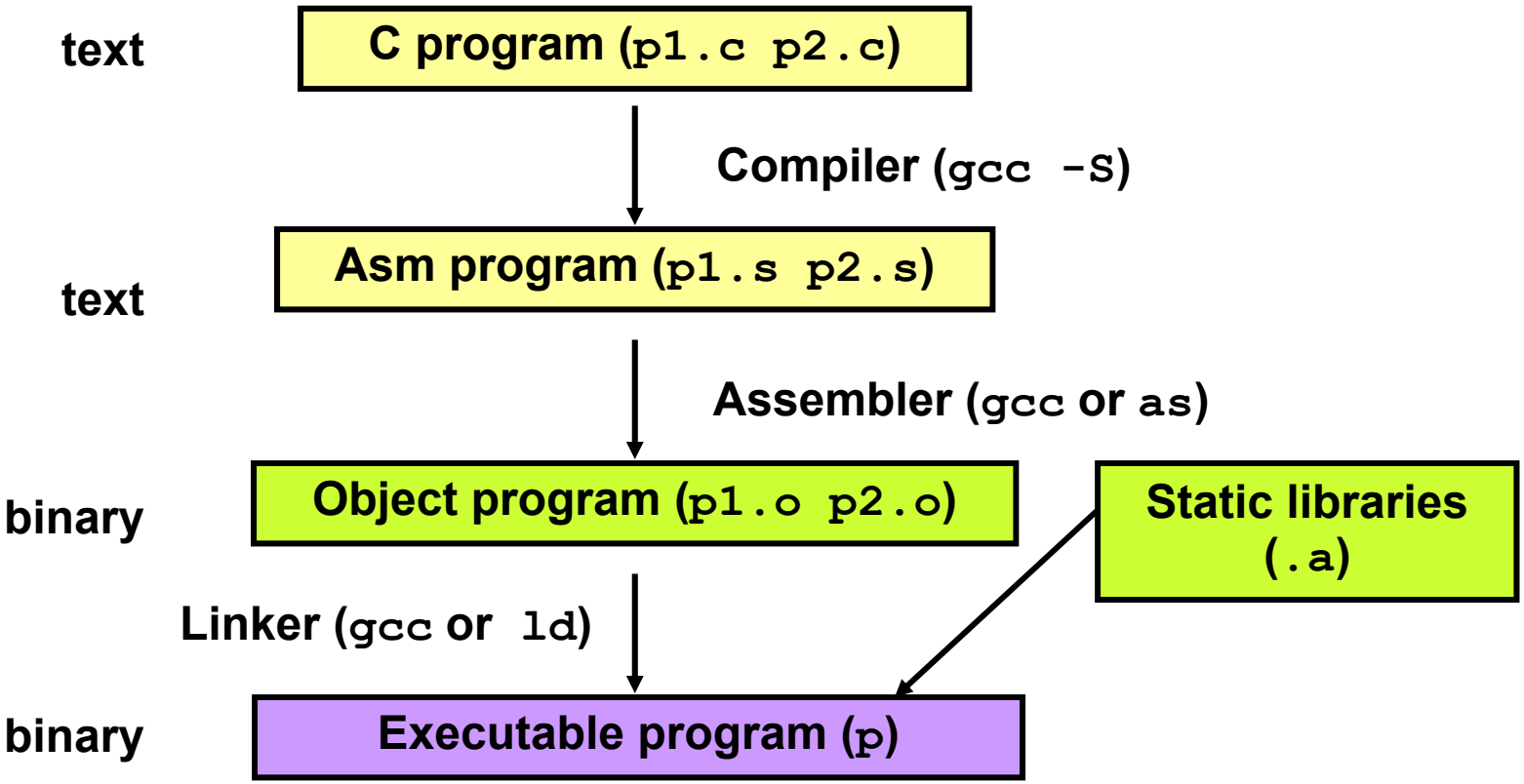
popl %edx





Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O p1.c p2.c -o p`



Compiling Into Assembly

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated Assembly

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

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`

Assembly Characteristics

- Minimal Data Types
  - "Integer" data of 1, 2, or 4 bytes
    - Data values
    - Addresses (untyped pointers)
  - Floating point data of 4, 8, or 10 bytes
  - No aggregate types such as arrays or structures
    - Just contiguously allocated bytes in memory
- 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

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`**

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for `malloc`, `printf`
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

Machine Instruction Example

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Similar to expression

`x += y`

a file
from
the
registe
r and
stores
back
into
%eax

x =
eax

```
0x401046:     03 45 08
```

- C Code
  - Add two signed integers
- Assembly
  - Add two 4-byte integers
    - "Long" words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:

`x`:	Register	`%eax`
`y`:	Memory	M[`%ebp+8`]
`t`:	Register	`%eax`

    - Return function value in `%eax`
- Object Code
  - 3-byte instruction
  - Stored at address `0x401046`

Disassembling Object Code

Disassembled

```
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
   d:       8d 76 00            lea    0x0(%esi),%esi
```

- Disassembler

  `objdump -d p`

  – Useful tool for examining object code

  – Analyzes bit pattern of series of instructions

  – Produces approximate rendition of assembly code

  – Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly

Object

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

Disassembled

```
0x401040 <sum>:      push   %ebp
0x401041 <sum+1>:    mov    %esp,%ebp
0x401043 <sum+3>:    mov    0xc(%ebp),%eax
0x401046 <sum+6>:    add    0x8(%ebp),%eax
0x401049 <sum+9>:    mov    %ebp,%esp
0x40104b <sum+11>:   pop    %ebp
0x40104c <sum+12>:   ret
0x40104d <sum+13>:   lea    0x0(%esi),%esi
```

- Within gdb Debugger

  `gdb p`

  `disassemble sum`

  – Disassemble procedure

  `x/13b sum`

  – Examine the 13 bytes starting at `sum`

What Can be Disassembled?

```
% 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
```

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

Moving Data

- Moving Data

  `movl` *Source,Dest*:

  – Move 4-byte ("long") word

  – Lots of these in typical code

- Operand Types

  – Immediate: Constant integer data

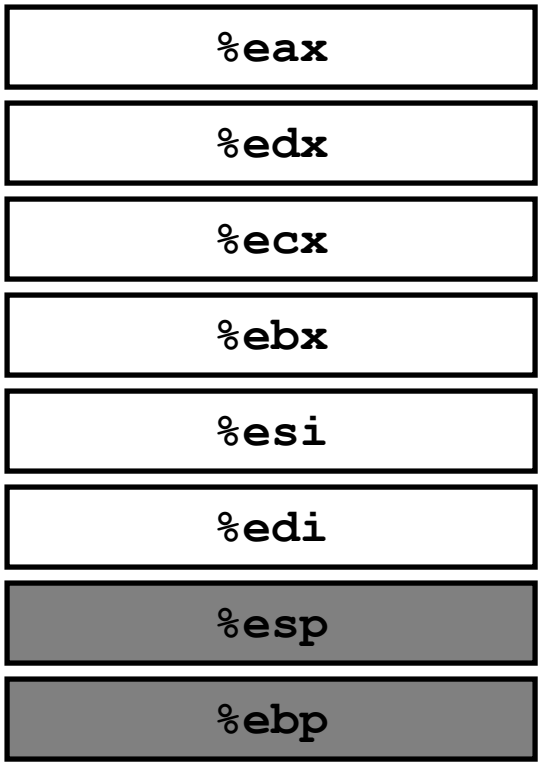
    - Like C constant, but prefixed with '`$`'
    - E.g., `$0x400`, `$-533`
    - Encoded with 1, 2, or 4 bytes

  – Register: One of 8 integer registers

    - But `%esp` and `%ebp` reserved for special use
    - Others have special uses for particular instructions

  – Memory: 4 consecutive bytes of memory

    - Various "address modes"



movl Operand Combinations

	Source	Destination		C Analog
movl	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

– Cannot do memory-memory transfers with single instruction

Simple Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address

```
movl (%ecx),%eax
```

- Displacement D(R)
  Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movl 8(%ebp),%edx
```

additio
n add
the
outside
to the
inside
this
would
be
useful
for an
array

Using Simple Addressing Modes

```
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
```

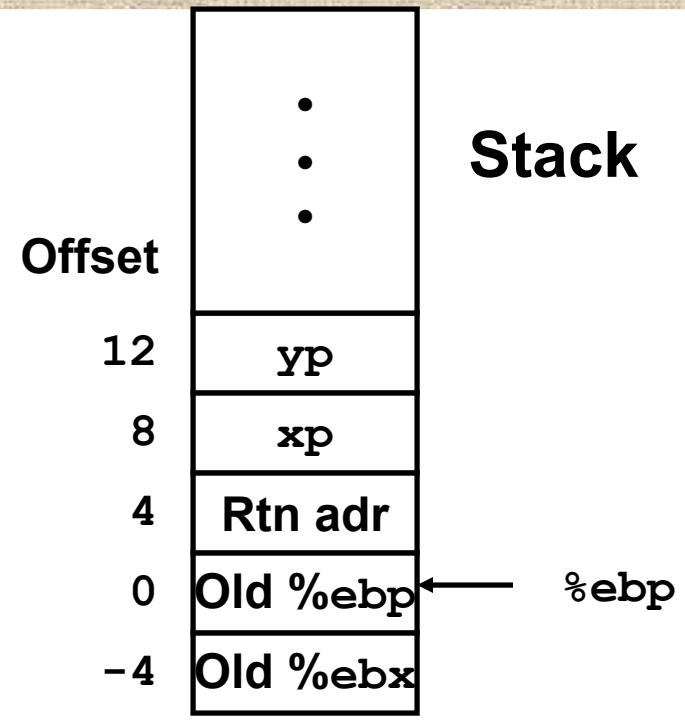
Set Up

Body

Finish

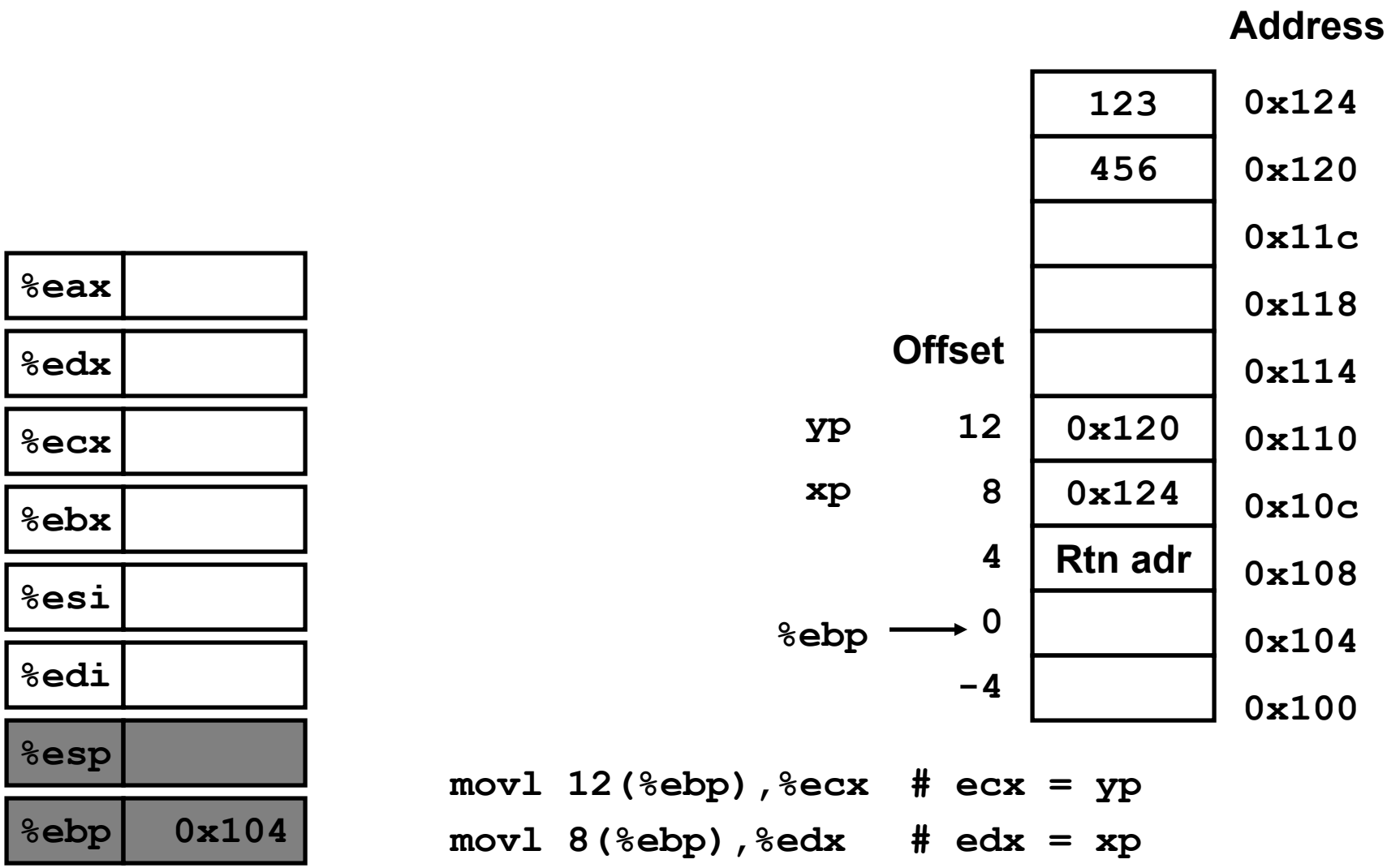
Understanding Swap

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

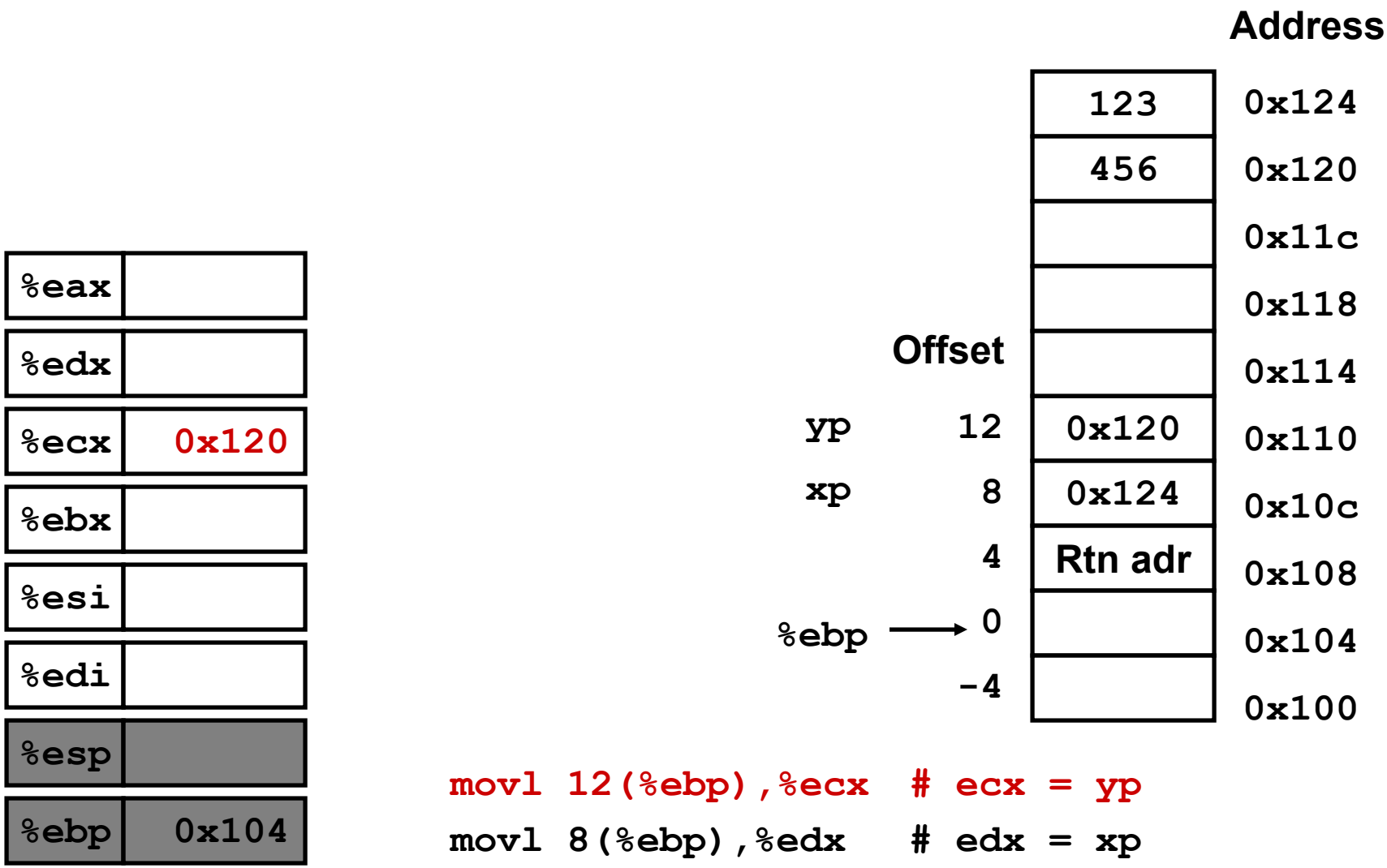


Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```




```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



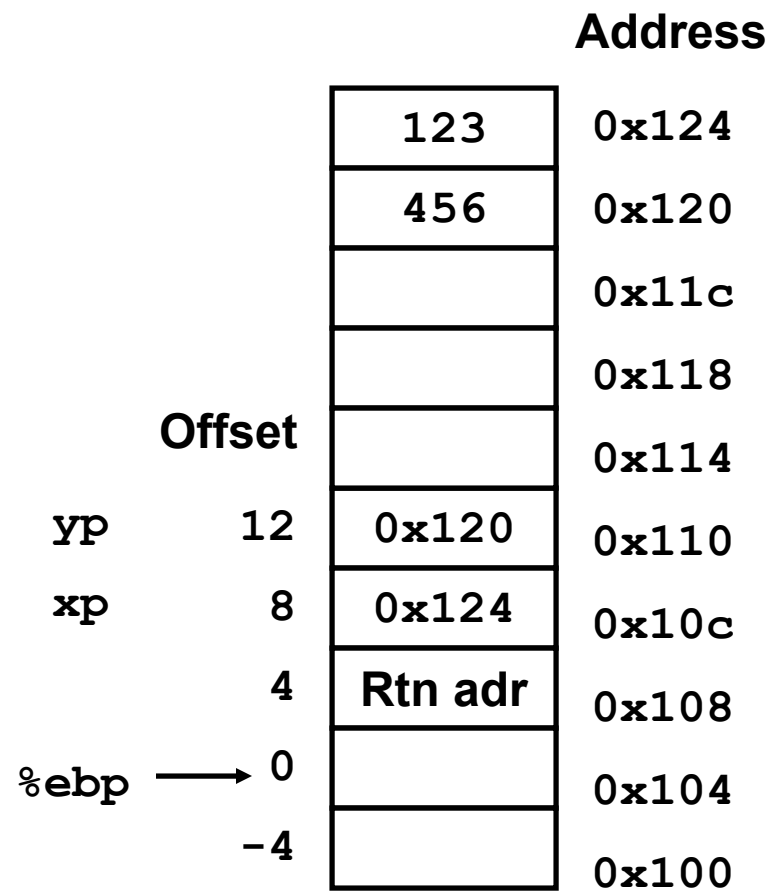

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104



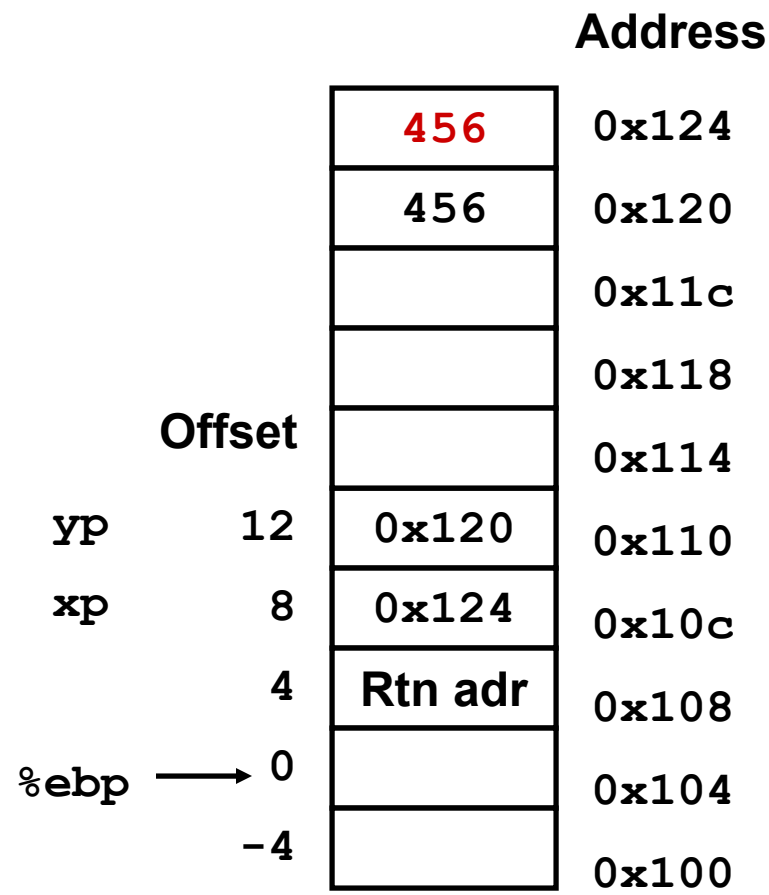

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

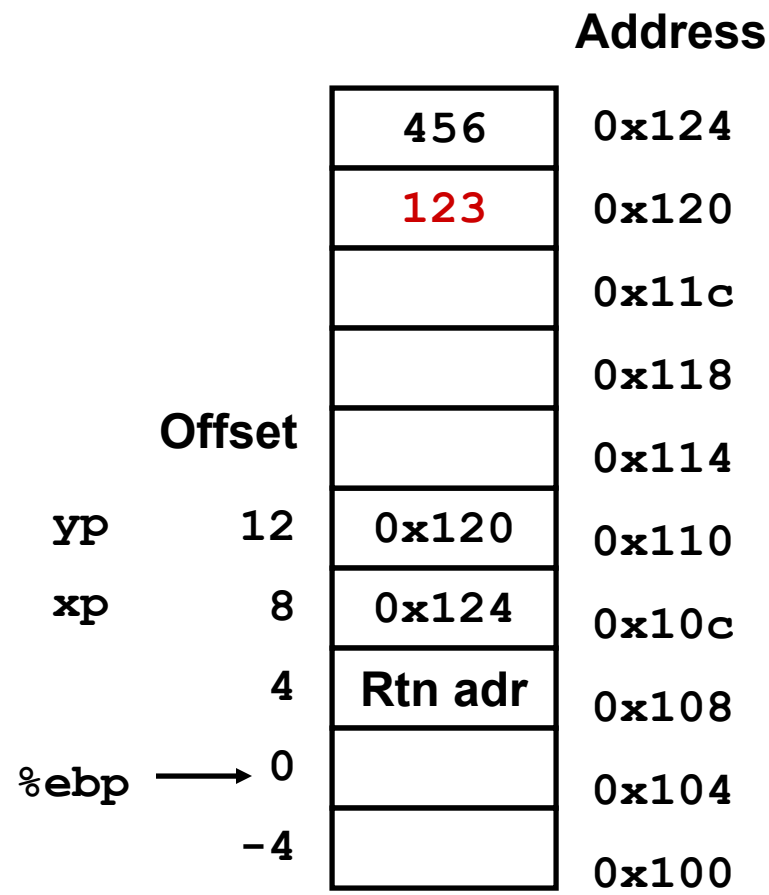
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Indexed Addressing Modes

- Most General Form

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

  – D: Constant "displacement" 1, 2, or 4 bytes

  – Rb: Base register: Any of 8 integer registers ebp

  – Ri: Index register: Any, except for `%esp`

    - Unlikely you'd use `%ebp`, either

  – S: Scale: 1, 2, 4, or 8

- Special Cases

  (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]

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

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

Address Computation Examples

%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

Address Computation Instruction

- `leal` *Src,Dest*
  - *Src* is address mode expression
  - Set *Dest* to address denoted by expression
- Uses
  - Computing address without doing memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8.

18(%eax), %ebx = it is for address calculation. It gets the memory without

Some Arithmetic Operations

Format		Computation	
• Two Operand Instructions			
`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*	

Some Arithmetic Operations

Format	Computation

- One Operand Instructions

`incl` *Dest*	*Dest* = *Dest* + 1
`decl` *Dest*	*Dest* = *Dest* – 1
`negl` *Dest*	*Dest* = – *Dest*
`notl` *Dest*	*Dest* = ~ *Dest*

More Arithmetic Operations

Format	Computation
`imull` *Source (signed)*	*R[%edx]:R[%eax] = Source x R[%eax]*
`mull` *Source (unsigned)*	*R[%edx]:R[%eax] = Source x R[%eax]*
`cltd`	*R[%edx]:R[%eax] = SignExtend(R[%eax])*

takes a 32 bit value and sign

Using leal for Arithmetic Expressions

```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

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

    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax

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

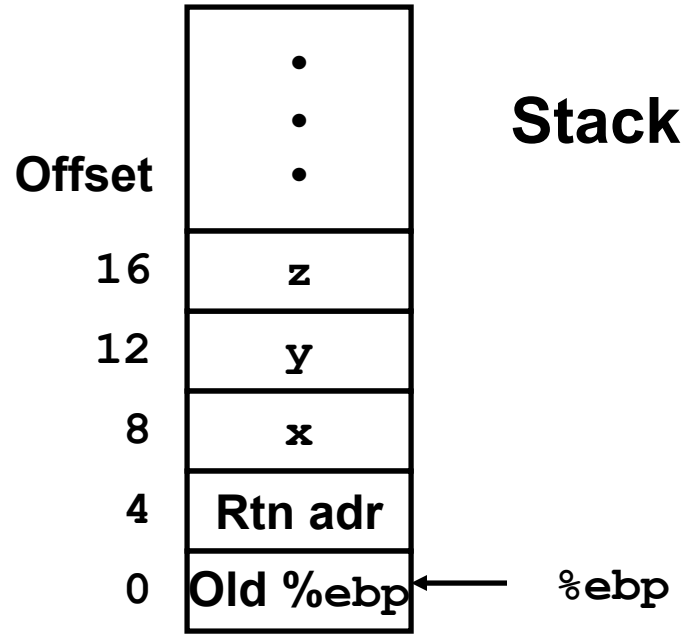
Set Up

Body

Finish

Understanding arith

```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y  (t1)
leal (%edx,%edx,2),%edx    # edx = 3*y
sall $4,%edx               # edx = 48*y (t4)
addl 16(%ebp),%ecx         # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax            # eax = t5*t2 (rval)
```

Understanding arith

```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

```
# eax = x
  movl 8(%ebp),%eax
# edx = y
  movl 12(%ebp),%edx
# ecx = x+y  (t1)
  leal (%edx,%eax),%ecx
# edx = 3*y
  leal (%edx,%edx,2),%edx
# edx = 48*y (t4)
  sall $4,%edx
# ecx = z+t1 (t2)
  addl 16(%ebp),%ecx
# eax = 4+t4+x (t5)
  leal 4(%edx,%eax),%eax
# eax = t5*t2 (rval)
  imull %ecx,%eax
```

Another Example

```
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

$2^{13} = 8192$, $2^{13} - 7 = 8185$

```
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
```

CISC Properties

- Instruction can reference different operand types
  - Immediate, register, memory
- Arithmetic operations can read/write memory
- Memory reference can involve complex computation
  - Rb + S*Ri + D
  - Useful for arithmetic expressions, too
- Instructions can have varying lengths
  - IA32 instructions can range from 1 to 15 bytes

Whose Assembler?

Intel/Microsoft Format

```
lea   eax,[ecx+ecx*2]
sub   esp,8
cmp   dword ptr [ebp-8],0
mov   eax,dword ptr [eax*4+100h]
```

GAS/Gnu Format

```
leal (%ecx,%ecx,2),%eax
subl $8,%esp
cmpl $0,-8(%ebp)
movl $0x100(,%eax,4),%eax
```

- Intel/Microsoft Differs from GAS
  - Operands listed in opposite order

    `mov` Dest, Src `movl` Src, Dest
  - Constants not preceded by '$', Denote hex with 'h' at end

    `100h` `$0x100`
  - Operand size indicated by operands rather than operator suffix

    `sub` `subl`
  - Addressing format shows effective address computation

    `[eax*4+100h]` `$0x100(,%eax,4)`

Data Types in x86-64

C declaration	Intel data type	GAS suffix	x86-64 Size (Bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`unsigned`	Double word	`l`	4
`long int`	Quad word	`q`	8
`unsigned long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`d`	8
`long double`	Extended precision	`t`	16

x86-64 Registers

rax rbp, the r in front says it represe ntedd as a 64 bit.

63	31		15	8 7	0	
%rax	%eax	%ax	%ah		%al	Return value
%rbx	%ebx	%bx	%bh		%bl	Callee saved
%rcx	%ecx	%cx	%ch		%cl	4th argument
%rdx	%edx	%dx	%dh		%dl	3rd argument
%rsi	%esi	%si			%sil	2nd argument
%rdi	%edi	%di			%dil	1st argument
%rbp	%ebp	%bp			%bpl	Callee saved
%rsp	%esp	%sp			%spl	Stack pointer
%r8	%r8d	%r8w			%r8b	5th argument
%r9	%r9d	%r9w			%r9b	6th argument
%r10	%r10d	%r10w			%r10b	**Caller Saved**
%r11	%r11d	%r11w			%r11b	**Caller Saved**
%r12	%r12d	%r12w			%r12b	**Callee Saved**
%r13	%r13d	%r13w			%r13b	Callee saved
%r14	%r14d	%r14w			%r14b	Callee saved
%r15	%r15d	%r15w			%r15b	Callee saved

Swap in 32-bit Mode

```
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
```

Setup

Body

Finish

Swap in 64-bit Mode

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

```
swap:
    movl    (%rdi), %edx
    movl    (%rsi), %eax
    movl    %eax, (%rdi)
    movl    %edx, (%rsi)
    retq
```

- Operands passed in registers (why useful?)
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required
- 32-bit data
  - Data held in registers %eax and %edx
  - movl operation

Swap Long Ints in 64-bit Mode

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

```
swap_l:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    retq
```

- 64-bit data
  - Data held in registers **`%rax`** and **`%rdx`**
  - **`movq`** operation
  - "q" stands for quad-word

gdb

```
 40046b:       c9                      leaveq
 40046c:       c3                      retq
 40046d:       0f 1f 00                nopl   (%rax)

0000000000400470 <frame_dummy>:
 400470:       55                      push   %rbp
 400471:       48 83 3d 37 02 20 00    cmpq   $0x0,2097719(%rip)        # 6006b
<__JCR_END__>
 400478:       00
 400479:       48 89 e5                mov    %rsp,%rbp
 40047c:       74 16                   je     400494 <frame_dummy+0x24>
 40047e:       b8 00 00 00 00          mov    $0x0,%eax
 400483:       48 85 c0                test   %rax,%rax
 400486:       74 0c                   je     400494 <frame_dummy+0x24>
 400488:       bf b0 06 60 00          mov    $0x6006b0,%edi
 40048d:       49 89 c3                mov    %rax,%r11
 400490:       c9                      leaveq
 400491:       41 ff e3                jmpq   *%r11
 400494:       c9                      leaveq
 400495:       c3                      retq
 400496:       90                      nop
 400497:       90                      nop
 400498:       90                      nop
 400499:       90                      nop
 40049a:       90                      nop
 40049b:       90                      nop
 40049c:       90                      nop
 40049d:       90                      nop
 40049e:       90                      nop
 40049f:       90                      nop

00000000004004a0 <main>:
 4004a0:       48 89 5c 24 f0          mov    %rbx,0xfffffffffffffff0(%rsp)
 4004a5:       4c 89 64 24 f8          mov    %r12,0xfffffffffffffff8(%rsp)
 4004aa:       48 83 ec 18             sub    $0x18,%rsp
 4004ae:       e8 f5 fe ff ff          callq  4003a8 <rand@plt>
 4004b3:       89 c3                   mov    %eax,%ebx
 4004b5:       e8 ee fe ff ff          callq  4003a8 <rand@plt>
 4004ba:       41 89 c4                mov    %eax,%r12d
 4004bd:       41 21 dc                and    %ebx,%r12d
 4004c0:       e8 e3 fe ff ff          callq  4003a8 <rand@plt>
 4004c5:       41 31 c4                xor    %eax,%r12d
 4004c8:       48 8b 5c 24 08          mov    0x8(%rsp),%rbx
 4004cd:       44 89 e0                mov    %r12d,%eax
 4004d0:       4c 8b 64 24 10          mov    0x10(%rsp),%r12
 4004d5:       48 83 c4 18             add    $0x18,%rsp
 4004d9:       c3                      retq
 4004da:       90                      nop
 4004db:       90                      nop
 4004dc:       90                      nop
 4004dd:       90                      nop
 4004de:       90                      nop
 4004df:       90                      nop

00000000004004e0 <__libc_csu_fini>:
 4004e0:       f3 c3                   repz retq
 4004e2:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)
 4004e9:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

00000000004004f0 <__libc_csu_init>:
 4004f0:       4c 89 64 24 e0          mov    %r12,0xffffffffffffffe0(%rsp)
 4004f5:       4c 89 6c 24 e8          mov    %r13,0xffffffffffffffe8(%rsp)
 4004fa:       4c 8d 25 8b 01 20 00    lea    2097547(%rip),%r12        # 60068
<__fini_array_end>
 400501:       4c 89 74 24 f0          mov    %r14,0xfffffffffffffff0(%rsp)
 400506:       4c 89 7c 24 f8          mov    %r15,0xfffffffffffffff8(%rsp)
 40050b:       49 89 f6                mov    %rsi,%r14
 40050e:       48 89 5c 24 d0          mov    %rbx,0xffffffffffffffd0(%rsp)
 400513:       48 89 6c 24 d8          mov    %rbp,0xffffffffffffffd8(%rsp)
 400518:       48 83 ec 38             sub    $0x38,%rsp
 40051c:       41 89 ff                mov    %edi,%r15d
 40051f:       49 89 d5                mov    %rdx,%r13
 400522:       e8 49 fe ff ff          callq  400370 <_init>
 400527:       48 8d 05 5e 01 20 00    lea    2097502(%rip),%rax        # 60068
<__fini_array_end>
 40052e:       49 29 c4                sub    %rax,%r12
 400531:       49 c1 fc 03             sar    $0x3,%r12
 400535:       4d 85 e4                test   %r12,%r12
 400538:       74 1e                   je     400558 <__libc_csu_init+0x68>
 40053a:       31 ed                   xor    %ebp,%ebp
 40053c:       48 89 c3                mov    %rax,%rbx
 40053f:       90                      nop
```

```
(gdb) break *0x4004c5
Breakpoint 3 at 0x4004c5
(gdb) run
Starting program: /w/fac.3/cs/reinman/code/a.out

Breakpoint 3, 0x00000000004004c5 in main ()
(gdb) i r
rax            0x643c9869       1681692777
rbx            0x6b8b4567       1804289383
rcx            0x31ead5106c     214393229420
rdx            0x31ead5107c     214393229436
rsi            0x7fffffffe974   140737488349556
rdi            0x31ead514c0     214393230528
rbp            0x0      0x0
rsp            0x7fffffffe990   0x7fffffffe990
r8             0x31ead51064     214393229412
r9             0x31ead510e0     214393229536
r10            0x0      0
r11            0x31eaa340e0     214389965024
r12            0x220b0146       571146566
r13            0x7fffffffea80   140737488349824
r14            0x0      0
r15            0x0      0
rip            0x4004c5 0x4004c5 <main+37>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
fctrl          0x37f    895
fstat          0x0      0
ftag           0xffff   65535
fiseg          0x0      0
fioff          0x0      0
foseg          0x0      0
fooff          0x0      0
fop            0x0      0
mxcsr          0x1f80   [ IM DM ZM OM UM PM ]
(gdb) si
0x00000000004004c8 in main ()
(gdb) i r
rax            0x643c9869       1681692777
rbx            0x6b8b4567       1804289383
rcx            0x31ead5106c     214393229420
rdx            0x31ead5107c     214393229436
rsi            0x7fffffffe974   140737488349556
rdi            0x31ead514c0     214393230528
rbp            0x0      0x0
rsp            0x7fffffffe990   0x7fffffffe990
r8             0x31ead51064     214393229412
r9             0x31ead510e0     214393229536
r10            0x0      0
r11            0x31eaa340e0     214389965024
r12            0x4637992f       1178048815
r13            0x7fffffffea80   140737488349824
r14            0x0      0
r15            0x0      0
rip            0x4004c8 0x4004c8 <main+40>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
fctrl          0x37f    895
fstat          0x0      0
ftag           0xffff   65535
fiseg          0x0      0
fioff          0x0      0
foseg          0x0      0
fooff          0x0      0
fop            0x0      0
mxcsr          0x1f80   [ IM DM ZM OM UM PM ]
(gdb) x/4x 0x4004c8
0x4004c8 <main+40>:     0x245c8b48      0xe0894408      0x24648b4c      0xc48348
10
(gdb) x/8x 0x4004c8
0x4004c8 <main+40>:     0x245c8b48      0xe0894408      0x24648b4c      0xc48348
10
0x4004d8 <main+56>:     0x9090c318      0x90909090      0x1f0fc3f3      0x000000
80
(gdb)
```