Assembly Language

Alan L. Cox
alc@rice.edu

Some slides adapted from CMU 15.213 slides

Objectives

Be able to read simple x86-64 assembly language programs

Why Learn Assembly Language?

- You'll probably never write a program in assembly
  - Compilers are much better and more patient than you are
- *But*, understanding assembly is key to understanding the machine-level execution model
  - Behavior of programs in presence of bugs
    - High-level language model breaks down
  - Tuning program performance
    - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state

Assembly Language

One assembly instruction

- Straightforward translation to a group of machine language bits that describe one instruction

What do these instructions do?

- Same kinds of things as high-level languages!
  - Arithmetic & logic
    - Core computation
  - Data transfer
    - Copying data between memory locations and/or registers
  - Control transfer
    - Changing which instruction is next

Assembly Language (cont.)

But, assembly language has additional features:

- Distinguishes instructions & data
- Labels = names for program control points
- Pseudo-instructions = special directives to the assembler
- Macros = user-definable abbreviations for code & constants

Example C Program

main.c:

```
#include <stdio.h>

void
hello(char *name, int hour, int min)
{

    printf("Hello, %s, it's %d:%02d.",
        name, hour, min);
}

int
main(void)
{

    hello("Alan", 2, 55);
    return (0);
}
```

Run the command:

UNIX% clang -S main.c

Output a file named main.s containing the assembly code for main.c

C Compiler’s Output

```
        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello, %s, it's %d:%02d."
        .text
.globl hello
        .type   hello, @function
hello:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        subq    $16, %rsp
.LCFI2:
        movq    %rdi, -8(%rbp)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movl    -16(%rbp), %ecx
        movl    -12(%rbp), %edx
        movq    -8(%rbp), %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
```

```
        call    printf
        leave
        ret
.LFE2:
        .size   hello, .-hello
        .section        .rodata
.LC1:
        .string "Alan"
        .text
.globl main
        .type   main, @function
main:
.LFB3:
        pushq   %rbp
.LCFI3:
        movq    %rsp, %rbp
.LCFI4:
        movl    $55, %edx
        movl    $2, %esi
        movl    $.LC1, %edi
        call    hello
        movl    $0, %eax
<..snip..>
```

A Breakdown of the Output

```
        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello, %s, it's %d:%02d."
        .text
.globl hello
        .type   hello, @function
hello:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        subq    $16, %rsp
.LCFI2:
        movq    %rdi, -8(%rbp)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movl    -16(%rbp), %ecx
        movl    -12(%rbp), %edx
        movq    -8(%rbp), %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
```

```
        call    printf
        leave
        ret
.LFE2:
        .size   hello, .-hello
<..snip..>
```

Instructions,
Pseudo-Instructions,
& Label Definitions

Instructions: Opcodes

```
        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello, %s, it's %d:%02d."
        .text
.globl hello
        .type   hello, @function
hello:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        subq    $16, %rsp
.LCFI2:
        movq    %rdi, -8(%rbp)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movl    -16(%rbp), %ecx
        movl    -12(%rbp), %edx
        movq    -8(%rbp), %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
```

```
        call    printf
        leave
        ret
.LFE2:
        .size   hello, .-hello
<..snip..>
```

Arithmetic,
data transfer, &
control transfer

Instructions: Operands

```
        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello, %s, it's %d:%02d."
        .text
.globl hello
        .type   hello, @function
hello:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        subq    $16, %rsp
.LCFI2:
        movq    %rdi, -8(%rbp)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movl    -16(%rbp), %ecx
        movl    -12(%rbp), %edx
        movq    -8(%rbp), %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
```

```
        call    printf
        leave
        ret
.LFE2:
        .size   hello, .-hello
<..snip..>
```

Registers, constants, & labels

Instruction Set Architecture

Contract between programmer and the hardware

- Defines visible state of the system
- Defines how state changes in response to instructions

Assembly Programmer (compiler)

- ISA is model of how a program will execute

Hardware Designer

- ISA is formal definition of the correct way to execute a program

Architecture vs. Implementation

Instruction Set Architecture

- Defines what a computer system does in response to a program and a set of data
- Programmer visible elements of computer system

Implementation

- Defines how a computer does it
- Sequence of steps to complete operations
- Time to execute each operation
- Hidden “bookkeeping” functions

Often Many Implementations of an ISA

ISA	Implementations
x86-64	Intel Core i7
	AMD FX-83XX
	VIA Nano
ARMv7-A	Apple A6/A6X
	Qualcomm Krait

Why separate architecture and implementation?

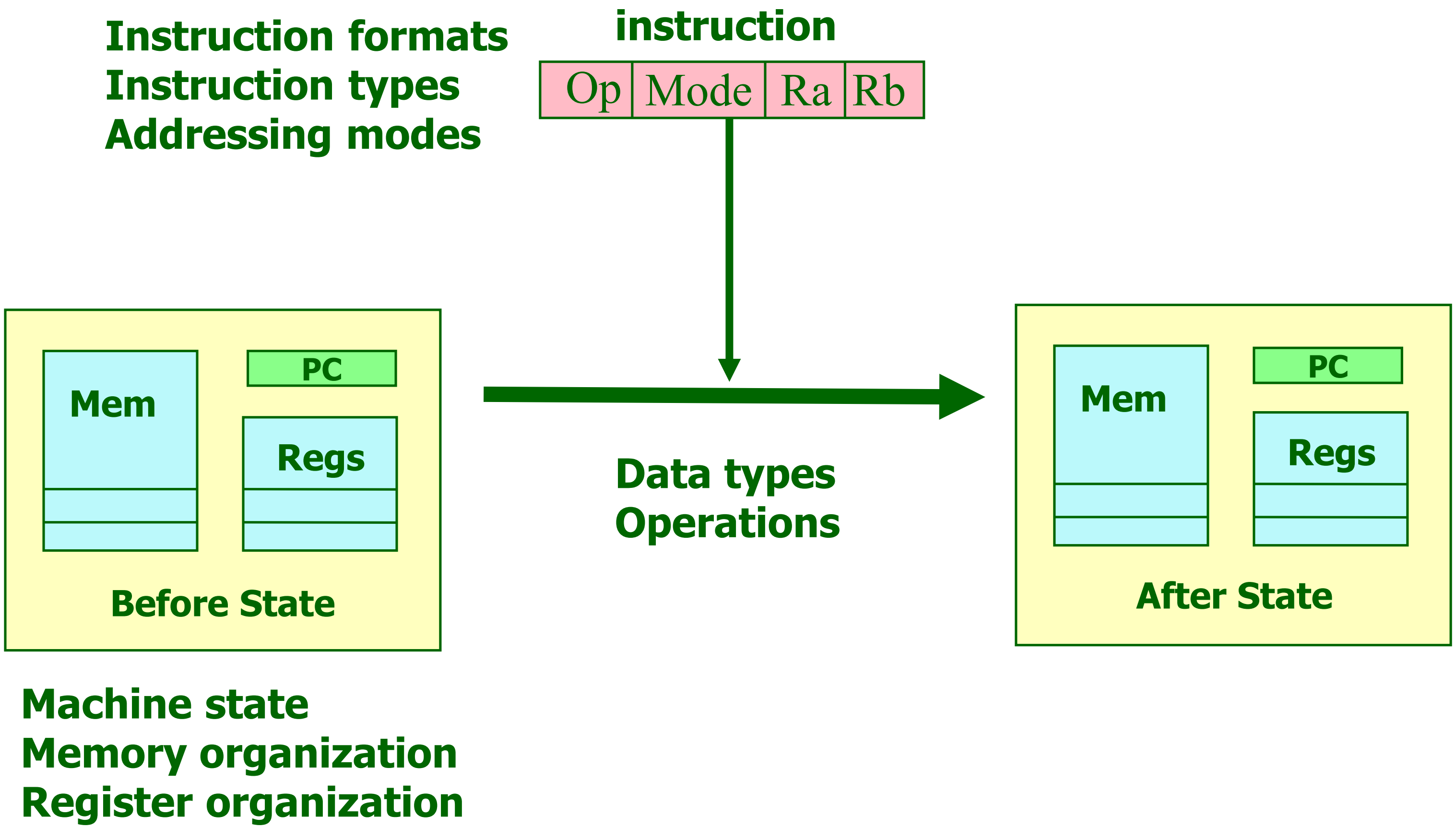
Compatibility

- VAX architecture: mainframe ⇒ single chip
- ARM: 20x performance range
  - high vs. low performance, power, price

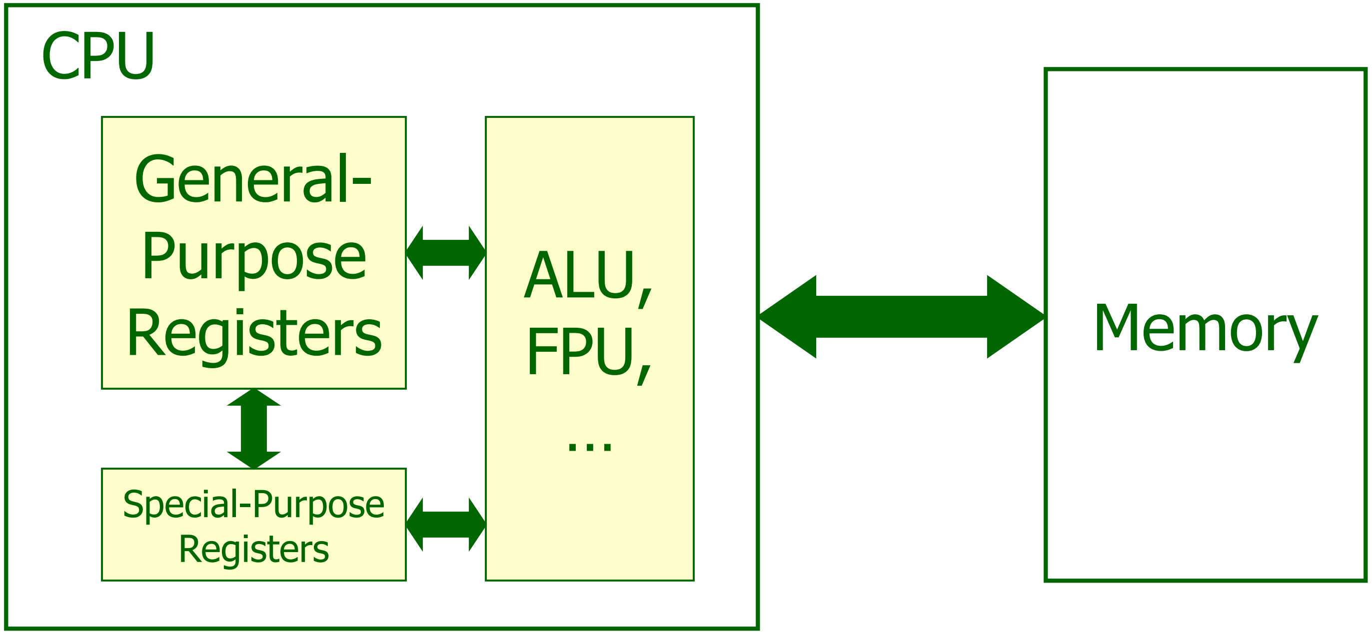
Longevity

- 20-30 years of ISA
- x86/x86-64 in 10th generation of implementations (architecture families)
- Retain software investment
- Amortize development costs over multiple markets

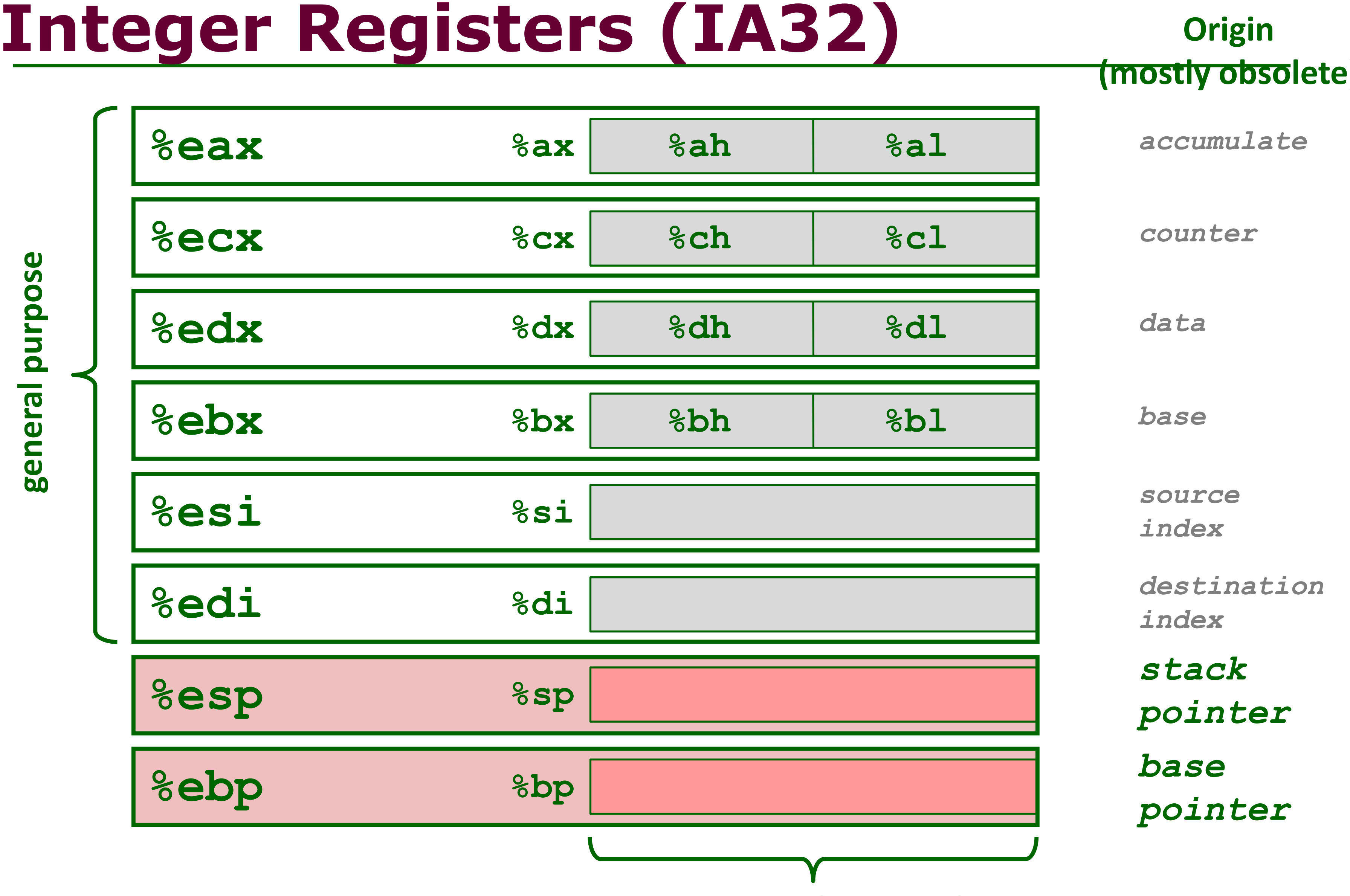
Instruction Set Basics



Typical Machine State



Integer Registers (IA32)



Moving Data: IA32

`%eax`
`%ecx`
`%edx`
`%ebx`
`%esi`
`%edi`
`%esp`
`%ebp`

Moving Data

`movl` *Source*, *Dest*:

Operand Types

- *Immediate:* **Constant integer data**
  - **Example:** `$0x400`, `$-533`
  - **Like C constant, but prefixed with** ‘`$`’
  - **Encoded with 1, 2, or 4 bytes**
- *Register:* **One of 8 integer registers**
  - **Example:** `%eax`, `%edx`
  - **But** `%esp` **and** `%ebp` **reserved for special use**
  - **Others have special uses for particular instructions**
- *Memory:* **4 consecutive bytes of memory at address given by register**
  - **Simplest example:** `(%eax)`
  - **Various other “address modes”**

`movl` Operand Combinations

	Source	Dest	Src,Dest	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 transfer with a single instruction

Simple Memory 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
```

Using Simple Addressing Modes

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

  popl  %ebx
  popl  %ebp
  ret
```

Set Up

Body

Finish

Using Simple Addressing Modes

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

   popl  %ebx
   popl  %ebp
   ret
```

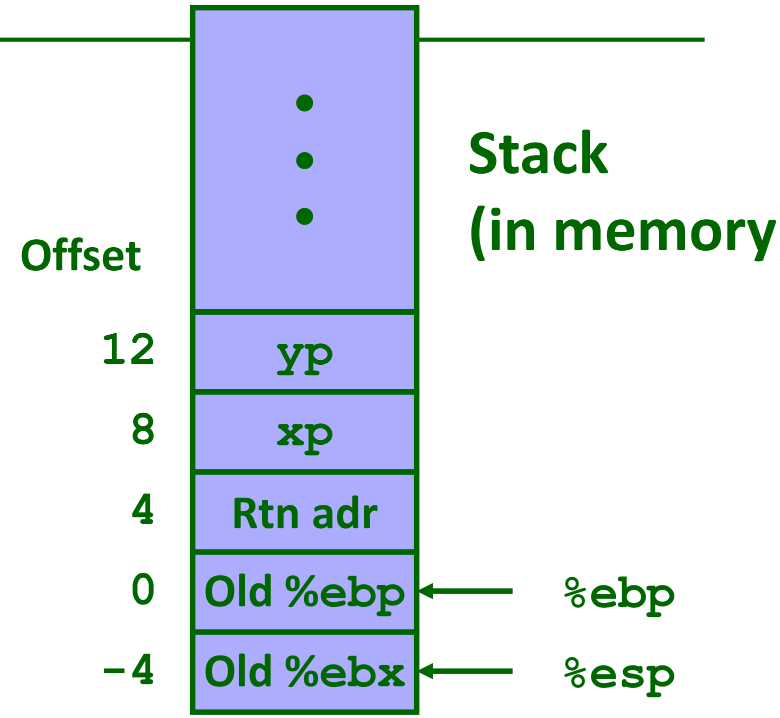
Set Up

Body

Finish

Understanding Swap

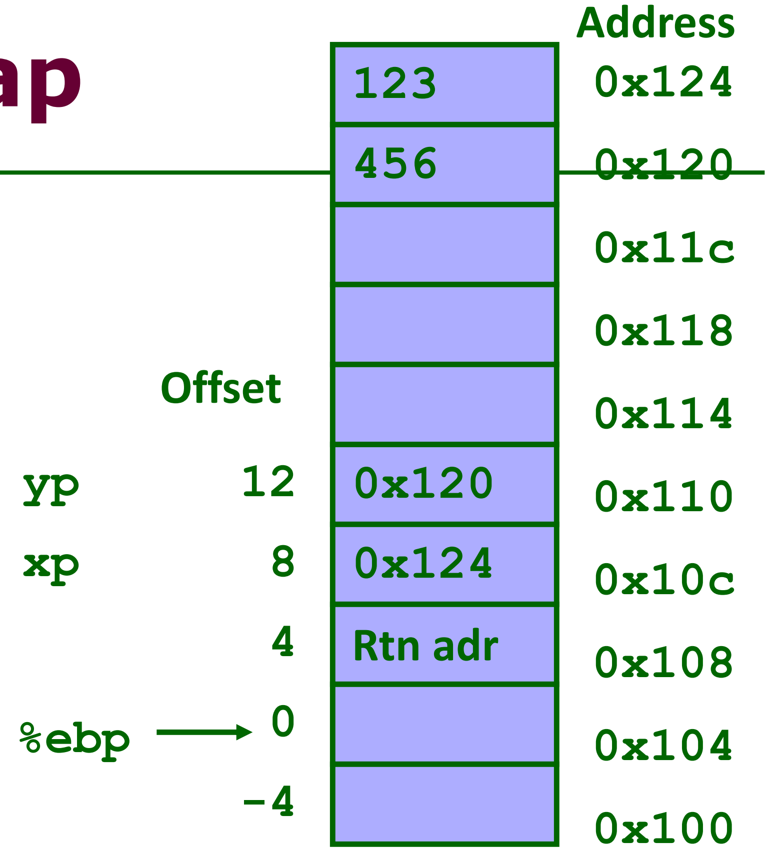
```
void
swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;

    *xp = t1;
    *yp = t0;
}
```



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

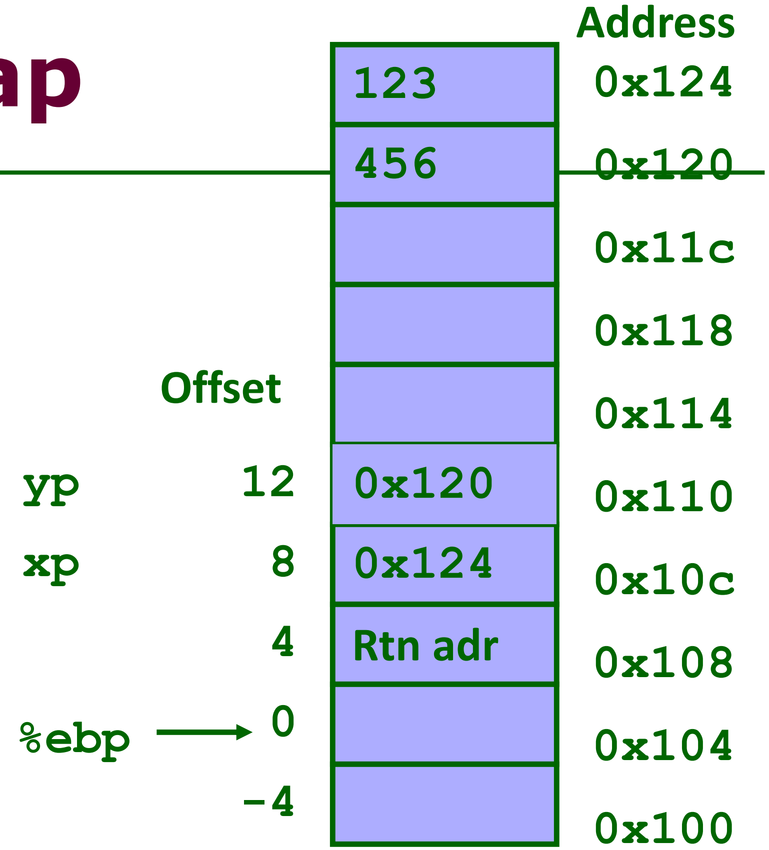
```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

Understanding Swap




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

Understanding Swap




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

Understanding Swap

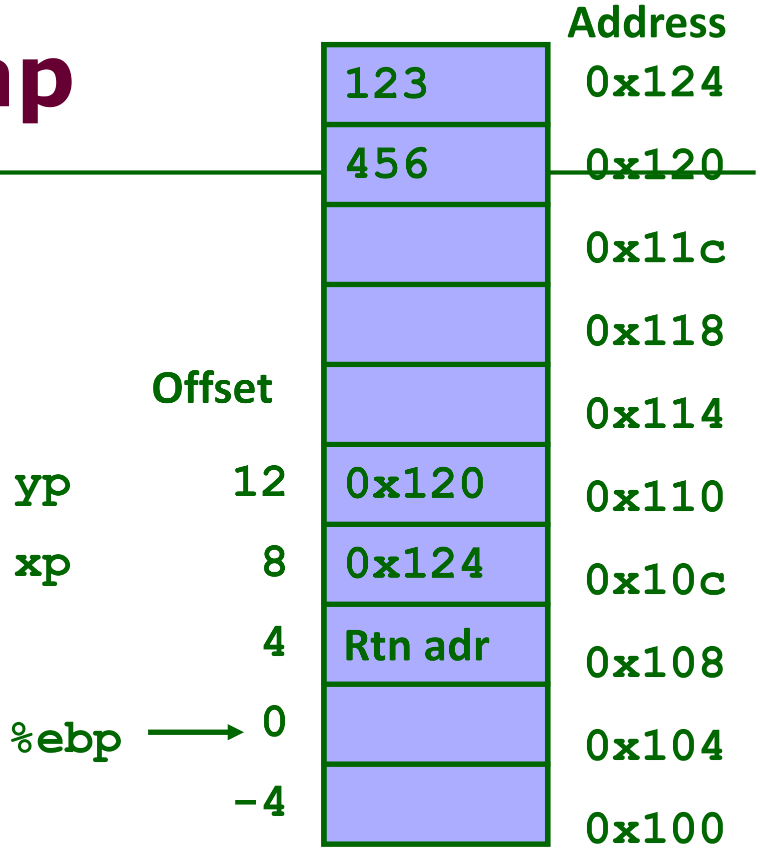
Register	Value
%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

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

Understanding Swap

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




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

Understanding Swap

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

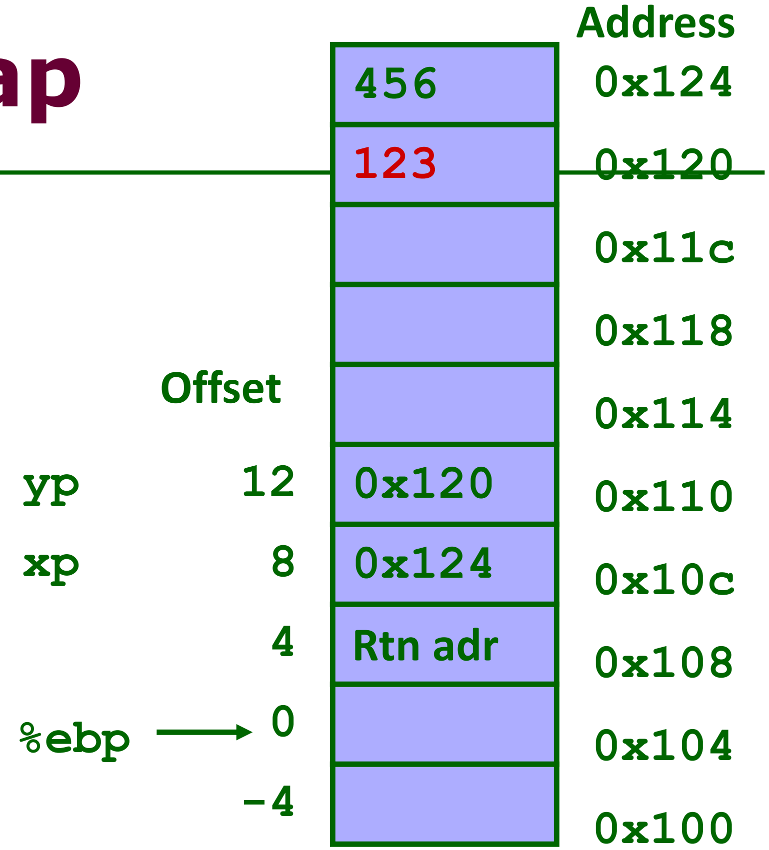
Understanding Swap

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

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

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

Understanding Swap



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

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

Complete Memory 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
- Ri: Index register: Any, except for `%esp`
  - Unlikely you'd use `%ebp`, either
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

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]]

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- **Extend existing registers. Add 8 new ones.**
- **Make `%ebp`/`%rbp` general purpose**

Instructions

Long word `l` (4 Bytes) ↔ Quad word `q` (8 Bytes)

New instructions:

- `movl` ➔ `movq`
- `addl` ➔ `addq`
- `sall` ➔ `salq`
- etc.

32-bit instructions that generate 32-bit results

- Set higher order bits of destination register to `0`
- Example: `addl`

32-bit code for swap

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

   popl   %ebx
   popl   %ebp
   ret
```

Set Up

Body

Finish

64-bit code for swap

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

  *xp = t1;
  *yp = t0;
}
```

```
swap:
                                   } Set Up

   movl    (%rdi), %edx            }
   movl    (%rsi), %eax            } Body
   movl    %eax, (%rdi)            }
   movl    %edx, (%rsi)            }

   ret                             } Finish
```

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

64-bit code for long int swap

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

  *xp = t1;
  *yp = t0;
}
```

```
swap_l:
                                   } Set Up

  movq    (%rdi), %rdx             }
  movq    (%rsi), %rax             } Body
  movq    %rax, (%rdi)             }
  movq    %rdx, (%rsi)             }

  ret                              } Finish
```

64-bit data

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

Application Binary Interface (ABI)

Standardizes the use of memory and registers by C compilers

- Enables interoperability of code compiled by different C compilers
  - E.g., a program compiled with Intel's optimizing C compiler can call a library function that was compiled by the GNU C compiler
- Sets the size of built-in data types
  - E.g., int, long, etc.
- Dictates the implementation of function calls
  - E.g., how parameters and return values are passed

Register Usage

The x86-64 ABI specifies that registers are used as follows

- **Temporary (callee can change these)**

  %rax, %r10, %r11

- **Parameters to function calls**

  %rdi, %rsi, %rdx, %rcx, %r8, %r9

- **Callee saves (callee can only change these after saving their current value)**

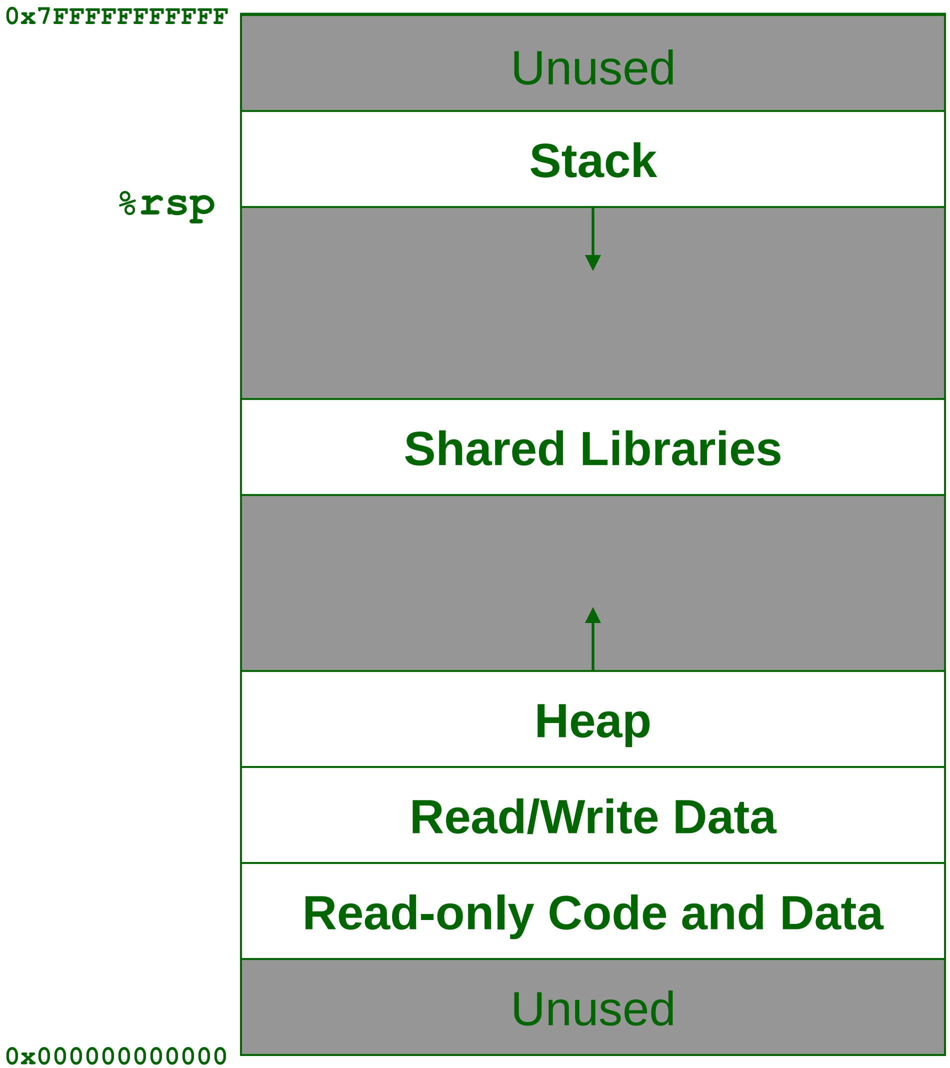
  %rbx, %rbp, %r12-%r15

  - %rbp is typically used as the "frame" pointer to the current function's local variables

- **Return values**

  %rax, %rdx

Procedure Calls and the Stack



Where are local variables stored?

- Registers (only 16)
- *Stack*

Stack provides as much local storage as necessary

- Until memory exhausted
- Each procedure allocates its own space on the stack

Referencing the stack

- `%rsp` points to the bottom of the stack in x86-64

Control Flow: Function Calls

What must assembly/machine language do?

Caller	Callee
1. Save function arguments 2. Branch to function body	3. Execute body • May allocate memory • May call functions 4. Save function result 5. Branch to where called

5. Use [illegible] location on stack)

Program Stack

Figure 3.3: Stack Frame with Base Pointer

Position	Contents	Frame
`8n+16(%rbp)`	memory argument eightbyte n	Previous
	...	
`16(%rbp)`	memory argument eightbyte 0	
`8(%rbp)`	return address	Current
`0(%rbp)`	previous `%rbp` value	
`-8(%rbp)`	unspecified	
	...	
`0(%rsp)`	variable size	
`-128(%rsp)`	red zone	

Figure 3.3 is reproduced from the AMD64 ABI Draft 0.99.5 by Matz et al.

What are Pseudo-Instructions?

Assembler directives, with various purposes

Data & instruction encoding:

- Separate instructions & data into sections
- Reserve memory with initial data values
- Reserve memory w/o initial data values
- Align instructions & data

Provide information useful to linker or debugger

- Correlate source code with assembly/machine
- …

Instructions & Pseudo-Instructions

```
        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello, %s, it's %d:%02d."
        .text
.globl hello
        .type   hello, @function
hello:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        subq    $16, %rsp
.LCFI2:
        movq    %rdi, -8(%rbp)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movl    -16(%rbp), %ecx
        movl    -12(%rbp), %edx
        movq    -8(%rbp), %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
```

```
        call    printf
        leave
        ret
.LFE2:
        .size   hello, .-hello
<..snip..>
```

Instructions,
Pseudo-Instructions,
& Label Definitions

Separate instructions & data

Instructions & Pseudo-Instructions

```
        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello, %s, it's %d:%02d."
        .text
.globl hello
        .type   hello, @function
hello:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        subq    $16, %rsp
.LCFI2:
        movq    %rdi, -8(%rbp)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movl    -16(%rbp), %ecx
        movl    -12(%rbp), %edx
        movq    -8(%rbp), %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
```

```
        call    printf
        leave
        ret
.LFE2:
        .size   hello, .-hello
<..snip..>
```

Instructions,
Pseudo-Instructions,
& Label Definitions

Reserve memory with initial data values

Instructions & Pseudo-Instructions

```
        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello, %s, it's %d:%02d."
        .text
.globl hello
        .type   hello, @function
hello:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        subq    $16, %rsp
.LCFI2:
        movq    %rdi, -8(%rbp)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movl    -16(%rbp), %ecx
        movl    -12(%rbp), %edx
        movq    -8(%rbp), %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
```

```
        call    printf
        leave
        ret
.LFE2:
        .size   hello, .-hello
<..snip..>
```

Instructions, Pseudo-Instructions, & Label Definitions

Correlate source code with assembly/machine

Label Types

```
        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello, %s, it's %d:%02d."
        .text
.globl hello
        .type   hello, @function
hello:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        subq    $16, %rsp
.LCFI2:
        movq    %rdi, -8(%rbp)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movl    -16(%rbp), %ecx
        movl    -12(%rbp), %edx
        movq    -8(%rbp), %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
```

```
        call    printf
        leave
        ret
.LFE2:
        .size   hello, .-hello
<..snip..>
```

Definitions,
internal references,
& external references

The label's value is the address of the subsequent instruction or pseudo-instruction

Assembly/Machine Language – Semantics

Basic model of execution

- Fetch instruction, from memory @ PC
- Increment PC
- Decode instruction
- Fetch operands, from registers or memory
- Execute operation
- Store result(s), in registers or memory

Simulate Program Execution

```
        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello, %s, it's %d:%02d."
        .text
.globl hello
        .type   hello, @function
hello:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        subq    $16, %rsp
.LCFI2:
        movq    %rdi, -8(%rbp)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movl    -16(%rbp), %ecx
        movl    -12(%rbp), %edx
        movq    -8(%rbp), %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
```

```
        call    printf
        leave
        ret
.LFE2:
        .size   hello, .-hello
        .section        .rodata
.LC1:
        .string "Alan"
        .text
.globl main
        .type   main, @function
main:
.LFB3:
        pushq   %rbp
.LCFI3:
        movq    %rsp, %rbp
.LCFI4:
        movl    $55, %edx
        movl    $2, %esi
        movl    $.LC1, %edi
        call    hello
        movl    $0, %eax
<..next slide..>
```

Simulate Program … (cont.)

```
        movl    $0, %eax
        leave
        ret
.LFE3:
        .size   main, .-main
<..snip..>
```

Exercise

0x700000000088
0x700000000084
0x700000000080
0x70000000007c
0x700000000078
0x700000000074
0x700000000070
0x70000000006c
0x700000000068
0x700000000064
0x700000000060
0x70000000005c
0x700000000058
0x700000000054
0x700000000050
0x70000000004c
0x700000000048
0x700000000044
0x700000000040
0x70000000003c
0x700000000038
0x700000000034
0x700000000030
0x70000000002c
0x700000000028
0x700000000024
0x700000000020
0x70000000001c
0x700000000018
0x700000000014
0x700000000010
0x70000000000c
0x700000000008
0x700000000004
0x700000000000

initial values:

%rbp	0x700000000088
%rsp	0x70000000006c
.LC0	0x408280
.LC1	0x408400

&"movl $0, %eax" in main() == 0x400220

%rbp

%rsp

%edi
%rdi

%esi
%rsi

%eax
%rax

%ecx
%rcx

%edx
%rdx

More x86-64 Assembly

- Chapter 3 of the textbook explains x86 and x86-64 assembly in greater detail
  - More examples translated from C
  - More detail than you're expected to know for this course
- Some code sequences generated by the compiler can still be confusing
  - Usually not important for this class (web is helpful if you are still curious)

Next Time

Program Linking