Assembly Language

Alan L. Cox

alc@cs.rice.edu

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**

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 Phenom II
	VIA Nano
SPARC V.9	UltraSPARC III
	HyperSPARC

Why separate architecture and implementation?

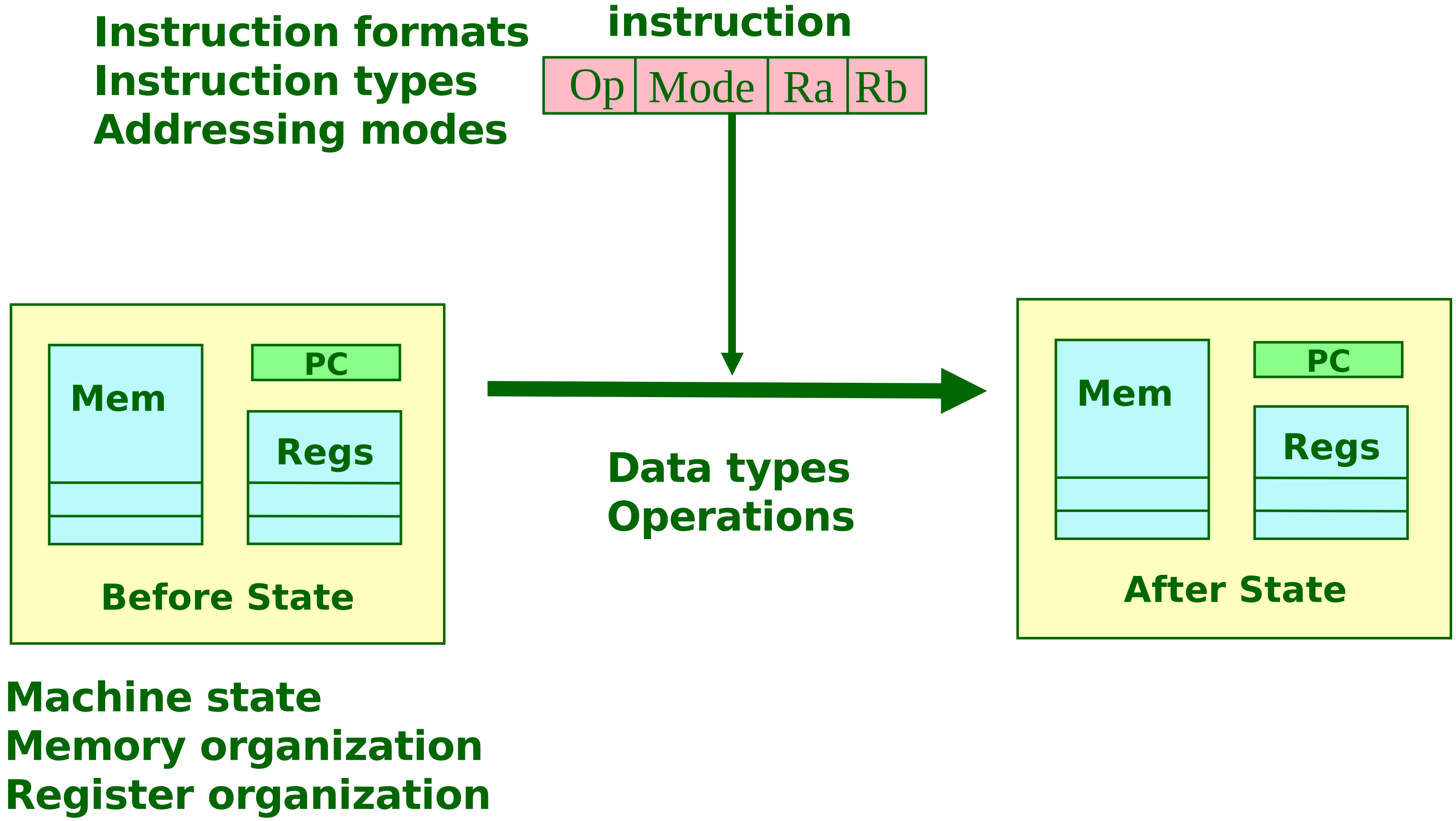
Compatibility

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

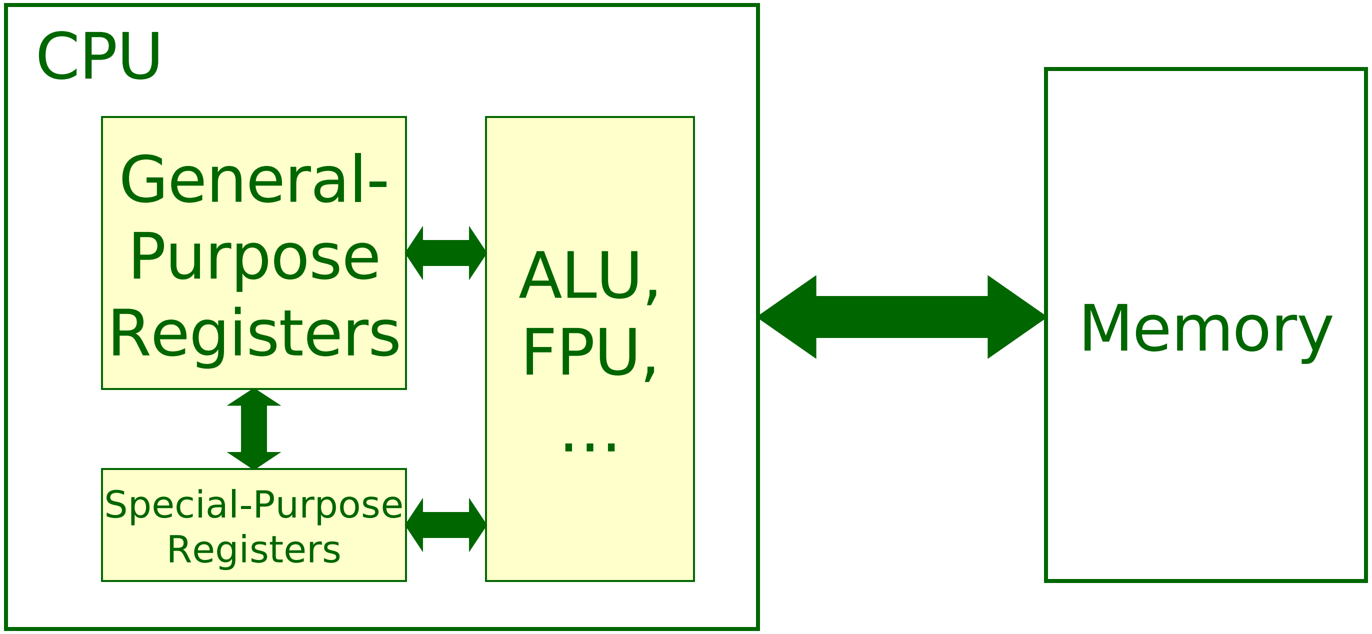
Longevity

- 20-25 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



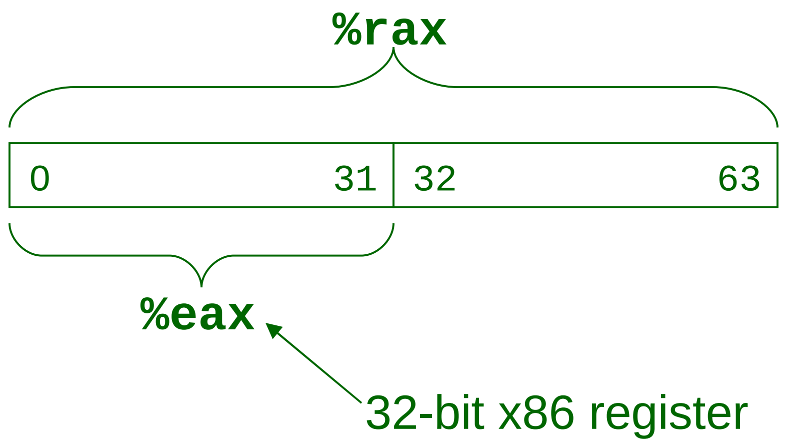
Typical Machine State



x86-64 Machine State

General-purpose registers

- **16 64-bit registers**
  - **Extended x86:** `%rax, %rbx, %rcx, %rdx, %rsp, %rbp, %rdi, %rsi`
  - **New:** `%r9 – %r15`
- **`%rsp` is always the stack pointer**
- **Backward compatibility for extended x86 registers**



x86-64 Machine State

Special-purpose registers

- **16 128-bit floating point registers**
  - **SSE: `%xmm0 - %xmm15`**
- **Each register stores two double precision or four single precision values**
  - **Vector arithmetic for speed!**

x86-64 Machine State

Memory

- **64-bit addresses**
  - However, only 48 bits used
    - 47 bits of address space for programs
    - 47 bits of address for operating system
- **Byte addressed**
- **Little-endian byte ordering within a word**
  - Least significant byte comes first in memory

Assembly Language

One assembly instruction

- One group of machine language bits

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

Instructions

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

Machine Language Encodings

Variable-width:

- 1 to dozens of bytes
- Flexible ISA
- CISC
- Harder hardware implementation
- x86/x86-64, VAX, …

Fixed-width:

- 1 word, typically
- Limited ISA
- RISC
- Easier hardware implementation
- SPARC, MIPS, PowerPC, …

Look at representative examples

Machine Language Instruction Formats

Different instructions have different kinds of operands

Can't encode all fields of all instructions within one word

- **Use different formats for different instructions**
- **Different architectures use different formats**
- **Different architectures group different sets of instructions**

SPARC:

- **Format 1: 1 30-bit immediate (call with immediate address)**
- **Format 2: 1 reg & 1 condition & 1 22-bit immediate (branches)**
- **Format 3: 3 regs, or 2 regs & 1 13-bit immediate (arithmetic, loads)**

Machine Language: Arithmetic

`add %o2,%o3,%o1`			0x9202800B
A variant of Format 3	op	math	10
	rd	%o1	01001
	op3	add	000000
	rs1	%o2	01010
	i	immediate?	0
	-	N/A	00000000
	rs2	%o3	01011

32 registers $\Rightarrow$ 5 bits

Not all bits important
Only used if i=1

Machine Language: Memory Ops

`ld [%o0+4], %o2`			0xD4022004
A variant of Format 3	op	memory	11
	rd	`%o2`	01010
	op3	`ld`	000000
	rs1	`%o0`	01000
	i	immediate?	1
	simm13	(13 bits)	0000000000100

Machine Language: Calls

```
         call there
         branch delay
         …
         …
there:   …
```

Instruction after branch/jump is *always* executed!

Assume assembler/linker assigns **there** to address 0x20000000

`call there`			0x48000000
Format 1	op	call	01
	disp30	(30 bits)	001000000000000000000000000000

`call` also saves the next program counter in %o7, which is used by `ret/retl` to return from a procedure (`ret` uses %i7, `retl` uses %o7)

Machine Language: Branches

	branch to **`dest`**
	branch delay
	instruction
`dest:`	destination instruction

3 instructions

`be dest`			0x02800003
Format 2	op	branch	00
	a	annul?	0
	cond	e	0001
	op2	b	010
	disp22	(22 bits)	0000000000000000000011

\# words relative to the branch instruction

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)**

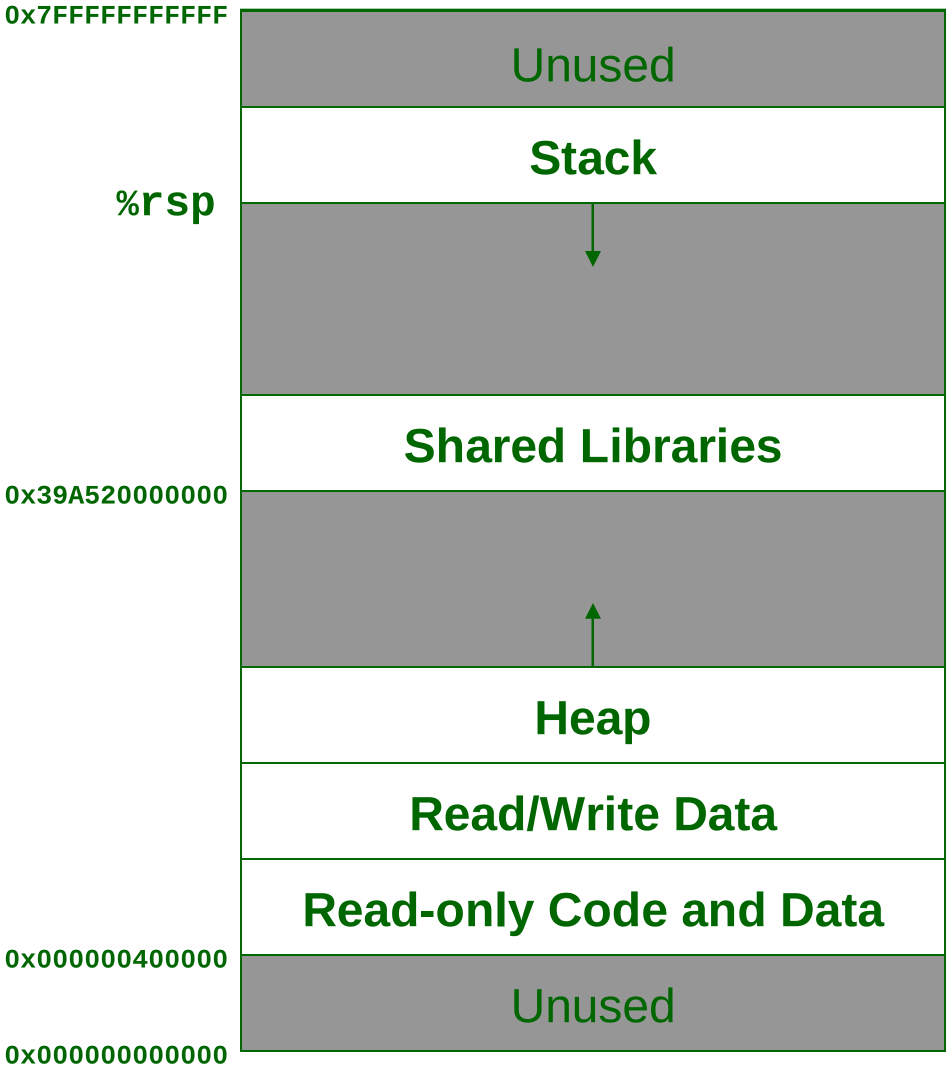
  %rbx, %rbp, %r12-%r15

  - %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

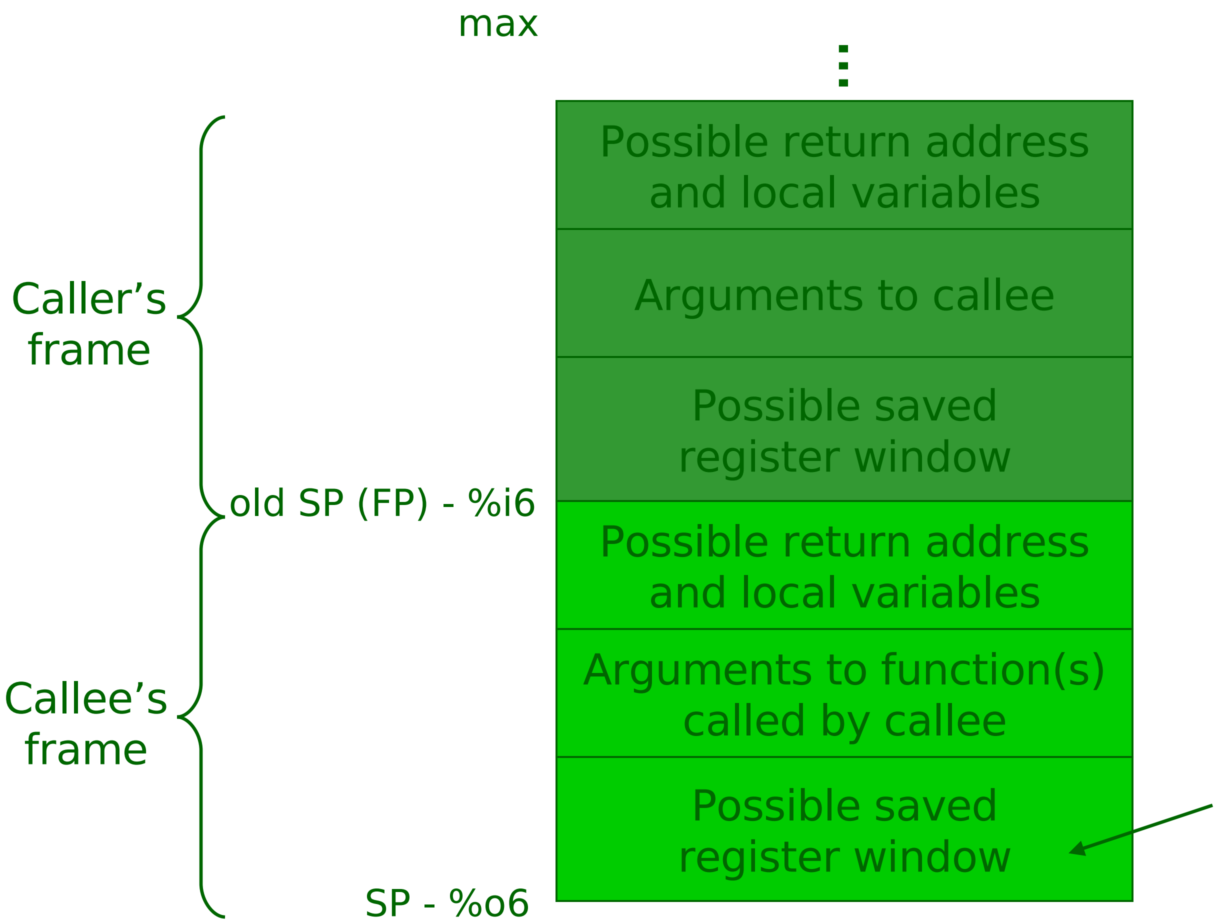
Control Flow: Function Calls

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

[illegible]

Program Stack



Misusing space allocated for local variable can trash return address!

Register windows are saved/restored under hardware control

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% gcc -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..>
```

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

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

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

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

Recall C Program

```
#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);
}
```

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..>
```

Program Execution (cont.)

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

More x86-64 Assembly

x86-64 notes on course web page

- Simple assembly language manual
- Some example code
- Along with lecture notes, should cover everything you need for the 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