Summary of Commands

Some of the commands we used on the first day. I do not expect you to remember them all.

- man
  - Display the manual page for the given command
  - E.g., man ls
  - Try "man man"
- sftp **student@simpson1 (mailto:student@simpson1)**
  - Transfer file from one computer to another
  - It accepts commands
    - help
    - ls
    - cd
    - get
  - ctrl-d to quit
- ls
  - List the files in the current directory
  - ls --help
- cat
  - Display the content of a file
- less
  - Display text one page at a time
  - Often used in combination with another program
  - E.g., "cat /usr/share/dict/words | less"
- od
  - Octal dump, output the content of a file in octal or hexadecimal
- objdump
  - Display the content of a binary or object file
- file
  - Determine the type of the given file
  - Try "file /etc/passwd", "file /boot/memtest86+.elf"
- Ctrl-Alt-F1, Ctrl-Alt-F2, ...
  - To switch among the consoles; only one has the graphics display

Speedup

The best way to express a performance improvement is a ratio called speedup.

For performance measured with time-based units (e.g., elapsed time in milliseconds or hours), this ratio is T_{old}/T_{new} where T_{old} is the time measured for the original version and T_{new} is the time measured for the modified (hopefully better) version. Speedup is unitless. Make sure both T_{old} and T_{new} are expressed with the same units.

For performance measured with rate-based units (e.g., frame per seconds), this ratio is R_{new}/R_{old} where R_{old} is the rate measured for the original version and R_{new} is the rate measured for the modified version.

Speedups greater than 1.0, means "new" is an improvement over "old". If the speedup is 1.2, we say "new" is 1.2 times better than "old". If the speedup is 2.0, we say "new" is 2 times better than "old".

Sometimes we report the improvement as a percentage. To do so, we must first remove 1.0 from the speedup. The result of this expression is the percentage improvement. For the two examples in the previous paragraph, the first "new" system is 20% faster the first "old" system and the second "new" system is 100% faster than the second "old" system.

Program Control: Recap

Condition Codes

In addition to integer registers, the CPU maintains a set of single-bit condition code registers. These registers can be tested to perform conditional branches. The most useful condition code registers are listed in the following table.

CC register	Name	Description
`CF`	Carry flag	The most recent operation generated a carry out of the most significant bit.
`ZF`	Zero flag	The most recent operation yielded 0.
`SF`	Sign flag	The most recent operation yielded a negative value.
`OF`	Overflow flag	The most recent operation caused a two's complement overflow (either negative or positive).

The *leaq* instruction does not alter any condition codes. Otherwise, all of the instructions listed in the ***Arithmetic and Logical Operations: Recap*** (https://canvas.instructure.com/courses/1517115/pages/arithmetic-and-logical-operations-recap) page cause the condition codes to be set. The logical operations clear *CF* and *OF* to 0, and set *ZF* and *SF* according to the result. The shift operations set *CF* to the last bit shifted out, set *ZF* and *SF* according to the result, and affect *OF* only for 1-bit shifts, otherwise it's undefined (I believe the book has got this part wrong). The *inc* and *dec* instructions set *SF*, *OF,* and *ZF* according to the result but leave *CF* unchanged.

In addition to the setting of condition codes by the instructions listed in the ***Arithmetic and Logical Operations: Recap*** (https://canvas.instructure.com/courses/1517115/pages/arithmetic-and-logical-operations-recap) , there are two instructions classes that set condition codes without altering any other registers.

Instruction	Based on	Description
CMP S_1, S_2	$S_2 - S_1$	Compare
`cmpb`		Compare byte
`cmpw`		Compare word

`cmpl`		Compare double word
`cmpq`		Compare quad word
TEST S_1, S_2	S_2 & S_1	Test
`testb`		Test byte
`testw`		Test word
`testl`		Test double word
`testq`		Test quad word

The *CMP* instructions set the condition codes according to the differences of their two operands. They set the condition codes in the same way as the corresponding *SUB* instructions, but *without* updating their destinations. With the assembler format we're using in this class, the operands are listed in reverse order, can make the code slightly awkward to read (i.e., the destination, which is also the first operand, appears on the right and the second operand appears on the left).

The *TEST* instructions set the condition codes in the same way as the corresponding *AND* instructions, but *without* updating their destinations. Typically, the same operand is repeated (e.g., *testq %rax, %rax* to see if *%rax* is negative, zero, or positive) or one operand is a mask indicating which bits should be tested.

Accessing the Condition Codes

Rather than reading the condition codes directly, there are three common ways of using the condition codes: (1) set a single byte to 0x00 or 0x01 depending on some combination of the condition codes, (2) jump conditionally to some other part of the program, or (3) transfer data conditionally.

Set a Single Byte

The *SET* instructions set a singe byte to 0x00 or 0x01 depending on some combination of the condition codes.

Instruction	**Synonym**	**Effect**	**Set condition**
`sete D`	`setz`	*D* ← *ZF*	Equal/zero
`setne D`	`setnz`	*D* ← *~ZF*	Not equal/not zero

`sets D`		*D* ← *SF*	Negative
`setns D`		*D* ← ~*SF*	Nonnegative
`setg D`	`setnle`	*D* ← ~(*SF* ^ *OF*) & ~*ZF*	Greater (signed >)
`setge D`	`setnl`	*D* ← ~(*SF* ^ *OF*)	Greater or equal (signed ≥)
`setl D`	`setnge`	*D* ← *SF* ^ *OF*	Less (signed <)
`setle D`	`setng`	*D* ← (*SF* ^ *OF*) & ~*ZF*	Less or equal (signed ≤)
`seta D`	`setnbe`	*D* ← ~*CF* & ~*ZF*	Above (unsigned >)
`setae D`	`setnb`	*D* ← ~*CF*	Above or equal (unsigned ≥)
`setb D`	`setnae`	*D* ← *CF*	Below (unsigned <)
`setbe D`	`setna`	*D* ← *CF* | *ZF*	Below or equal (unsigned ≤)

Jump Instructions

Under normal execution, instructions follow each other in the order they are listed. A *jump* instruction can cause the execution to switch to a completely new position in the program.

Instruction	Synonym	Jump condition	Description
`jmp Label`		1 (always)	Direct jump
`jmp *Operand`		1 (always)	Indirection jump
`je Label`	`jz`	*ZF*	Equal/zero
`jne Label`	`jnz`	~*ZF*	Not equal/not zero
`js Label`		*SF*	Negative

`jns Label`		*~SF*	Nonnegative
`jg Label`	`jnle`	~(*SF* ^ *OF*) & *~ZF*	Greater (signed >)
`jge Label`	`jnl`	~(*SF* ^ *OF*)	Greater or equal (signed ≥)
`jl Label`	`jnge`	*SF* ^ *OF*	Less (signed <)
`jle Label`	`jng`	(*SF* ^ *OF*) | *ZF*	Less or equal (signed ≤)
`ja Label`	`jnbe`	*~CF* & *~ZF*	Above (unsigned >)
`jae Label`	`jnb`	*~CF*	Above or equal (unsigned ≥)
`jb Label`	`jnae`	*CF*	Below (unsigned <)
`jbe Label`	`jna`	*CF* | *ZF*	Below or equal (unsigned ≤)

The first two *jump* instruction are unconditional. The first one is a direct *jump* where the destination is encoded as part of the instruction. The second one is an indirect *jump* where the destination is read from a register (e.g., *jmp *%rax*) or a memory location (e.g., *jmp *(%rax)*). The remaining *jump* instructions are conditional; they either jump or continue executing at the next instruction depending on some combination of the condition codes.

There are two different encodings of for the jump targets: (1) PC relative and (2) absolute. In PC relative, the instruction encoding specifies the difference between the address of the target instruction and the address of the instruction immediately *following* the jump (owing to how early Intel processors were implemented). The difference is encoded in two's complement. In absolute, the instruction encoding specifies the actual address of the target instruction. The assembler and linker select the appropriate encodings of the jump destinations.

Conditional Move Instructions

Instruction	Synonym	Move condition	Description
`cmove S, R`	`cmovz`	*ZF*	Equal/zero
`cmovne S, R`	`cmovnz`	*~ZF*	Not equal/not zero

`cmovs S, R`	`cmovz`	*SF*	Negative
`cmovns S, R`	`cmovz`	*~SF*	Nonnegative
`cmovg S, R`	`cmovnle`	~(*SF* ^ *OF*) & *~ZF*	Greater (signed >)
`cmovge S, R`	`cmovnl`	~(*SF* ^ *OF)*	Greater or equal (signed ≥)
`cmovl S, R`	`cmovnge`	*SF* ^ *OF*	Less (signed <)
`cmovle S, R`	`cmovng`	(*SF* ^ *OF*) | *ZF*	Less or equal (signed ≤)
`cmova S, R`	`cmovnbe`	*~CF* & *~ZF*	Above (unsigned >)
`cmovae S, R`	`cmovnb`	*~CF*	Above or equal (unsigned ≥)
`cmovb S, R`	`cmovnae`	*CF*	Below (unsigned <)
`cmovbe S, R`	`cmovna`	*CF* | *ZF*	Below or equal (unsigned ≤)

Integer Representation: Recap

Unsigned Integers

A memory location stores an unsigned integer as a binary encoding of the corresponding decimal number: 0 = 0_2, 1 = 1_2, 2 = 10_2, 3 = 11_2, 4 = 100_2, ..., 2^n-1 = 111...11_2 (a binary number with $n \times 1_2$), where n is the width of the number representation (typically 8, 16, 32, or 64 bits). Range 0 ... UMAX.

Signed Integers

Typically, but not always, stored as a two's complement number. (The C programming language standard does not say how a signed number should be encoded.) A positive number encoded the same an unsigned integer. A negative number encoded (typically) as a two's complement encoding:

Assume width of the number representation is n (e.g., 8, 16, 32, or 64 bits)
Take the number: −N (e.g., −1)
Remove the sign: N (+1 = 1_2)
Compute the complement: ~N (111...10_2, $n - 1 \times 1_2$ followed by 0_2)
Add one: ~N + 1 (111...10_2 + 1_2 = 111...11_2, $n \times 1_2$)

Range: −MAX − 1 ... MAX.

Why two's complement? Because the same adder (the piece of hardware that can compute the sum of two values) can be used for both unsigned and signed operands.

C Definitions

```
#include <limits.h>

SCHAR_MIN
SCHAR_MAX
UCHAR_MAX
SHRT_MIN
SHRT_MAX
USHRT_MAX
INT_MIN
INT_MAX
UINT_MAX
LONG_MAX
LONG_MIN
ULONG_MAX
```

Information Storage: Recap

Hexadecimal Notation

Decimal	0	1	2	3	4	5	6	7
Binary	0000	0001	0010	0011	0100	0101	0110	0111
Hexadecimal	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
Decimal	8	9	10	11	12	13	14	15
Binary	1000	1001	1010	1011	1100	1101	1110	1111
Hexadecimal	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF

Patterns:

- 2^n = $100...0_2$ (1 followed by n × 0s), e.g., 1, 2, 4, & 8.
- 2^n-1 = $11...1_2$ (n × 1s), e.g., 0, 1, 3, 7, & 15 (0xF).
- $11...10_2$ = 2 × $11...1_2$, (for the same strings of 1s), e.g., 6 (0110_2) = 2 × 3 (0011_2), 10 (1010_2) = 2 × 5 (0101_2), 14 (1110_2, 0xE) = 2 × 7 (0111_2), & 12 (1100_2, 0xC) = 2 × 6 (0110_2)

These patterns cover all hexadecimal digits except 5, 9, 0xB, & 0xD. But 5 = 4 + 1, 9 = 8 + 1, 0xB = 0xA + 1, & 0xD = 0xC + 1.

Representations

Bit patterns in memory or registers can represent:

- Integers
- Floating point numbers
- Characters
- Strings
- Machine instructions (i.e., code)
- Addresses
- Bit vectors

Types

C declaration	C declaration	Bytes
Signed	**Unsigned**	**x86_64**

char	unsigned char	1*
short	unsigned short	2
int	unsigned int	4
long	unsigned long	4
int32_t	uint32_t	4
int64_t	uint64_t	8
char *		8
float		4
double		8

Comments

- I have only shown the sizes for the machines on which we will do the labs.
- sizeof(type) is a C expression that returns the size of the given type in a number of bytes (e.g., sizeof(unsigned short)).
- Only the size of char is always going to be 1, the others will depend on the compiler, two different compilers on the same machine may choose different representations (i.e., sizes for the built-in types).
- The only thing the C standard promises is sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long).

Integers

- Finite, not very large, but precise
- Associative law holds
- Signed or unsigned
  - Two's complement representation

Floating Numbers

- Finite, much larger, but imprecise
- Associative law doesn't hold
- Representation like scientific notation (e.g., 6.022×10^{23})
  - Usually a sign, mantissa, and biased-exponent
  - Base 2 or 10
  - Also bit patterns reserved for $-\infty$, $+\infty$, and NaN (not a number, e.g., 0/0)

Characters

- char (or unsigned char) type
- ASCII character encoding
  - **https://en.wikipedia.org/wiki/ASCII** **(https://en.wikipedia.org/wiki/ASCII)**
  - man ascii
  - Originally based on the English alphabet (i.e., can't do Greek letters)
    - Unicode extends ASCII to include all other alphabets

Strings

- Array of characters terminated with the null (having value 0) character
- Always stored contiguously in memory

Machine Instructions

- Code executed by the microprocessor

Addresses

- For instance char *p
- Beware of pointer arithmetic

Bit vectors

- String of bits

Alignment

The compiler usually likes to assign an address to a variable according to its size (more precisely according to the size of the variable's base type). For instance, for a variable of a basic type, the compiler will assign an address that is a multiple of that type's size. Say, you declare "long l;", the compiler will almost always select an address for "l" that is a multiple of 8 (assuming the size of long is 8). When we talk about caches, we will understand the reason behind this behavior.

Byte ordering

A multi-byte programming object (e.g., a long) is usually stored in memory as a contiguous sequence of bytes, with the address of the object given by the smallest address of the bytes used.

For ordering the bytes representing an object, there are two common conventions:

- Little endian: store the object with the least significant byte first and most significant byte last.
- Big endian: store the object with the most significant byte first and least significant byte last.

The machines in our lab are based on the Intel/AMD processors which are little endian. IBM and Oracle processors are big endian. ARM can be either, although Android and IOS set it to little endian.

Byte ordering usually does not matter. It matters in the following cases:

1. Data communication over the network.
2. Looking at byte sequences representing integer data (e.g., output of the disassembler).
3. Code that circumvent the type system (e.g., casting).

Floating Point: Recap

Format

The IEEE floating point format consists of three fields (from right to left, or from least significant to most significant): fraction, exponent, and sign. For the two most common formats, these fields are packed in either a 32-bit (single-precision, "float") or 64-bit (double-precision, "double") words.

	Single-precision	**Double-precision**
Fraction width (*f*)	23 bits (bits 0-22)	52 bits (bits 0-51)
Exponent width (*e*)	8 bits (bits 23-30)	11 bits (bits 52-62)
Sign width (*s*)	1 bit (bit 31)	1 bit (bit 63)
Bias	127 (2^8-1)	1023 ($2^{11}-1$)

The interpretation of a given floating point representation depends on the value of the exponent field (*e*) (the table below assumes the single-precision floating point format; for double-precision 255 would become 2047, the rest stays the same).

Value of *e*	**Representation**	**Effective exponent (*E*)**	**Effective fraction (*M*)**	**Value**
$e = 0$	Denormalized	$E = 1 - \text{bias}$	$M = 0.f$	Value = $2^E \times M$
$0 < e < 255$	Normalized	$E = e - \text{bias}$	$M = 1.f$ (implied 1)	Value = $2^E \times M$
$e = 255$	Special values	Not applicable	Not applicable	Value = ∞ or NaN

Special values include ∞ and NaN. The latter stands for Not a Number and is used to encode results that cannot be given as a real number or infinity (e.g., 0/0, ∞ − ∞, or square root of −1).

Rounding

The IEEE floating point format specifies four rounding modes. The default mode is Round-to-even.

Mode	**$1.40**	**$1.60**	**$1.50**	**$2.50**	**$-1.50**
Round-to-even	$1	$2	$2	$2	$−2
Round-toward-zero	$1	$1	$1	$2	$−1
Round-down	$1	$1	$1	$2	$−2
Round-up	$2	$2	$2	$3	$−1

Floating Point in C

All versions of C provide two different floating-point data types: **float** and **double**. On machines that support IEEE floating point, these data types correspond to single- and double-precision floating point.

There is no standard method in C to change the rounding mode or to get the special values such as −0, +∞, −∞, or *NaN*.

Compilation System: Recap

We call a C program, like *hello.c*, a high-level program because it can be read, written, and understood by human beings. However, to run *hello.c* on a computing system the individual C statements comprising that program must be translated into a sequence of low-level machine-language instructions. These instructions are then packaged in an executable object program, executable for short, and stored as a binary disk file.

The compilation system, compiler for short, is in charge of transforming a high-level C program into a executable object program. It does so in 4 stages:

1. Preprocessing. The preprocessor (*cpp*) modifies the original C program according to directives that begin with the '#' character. The result is another C program stored in a file typically with the *.i* suffix.
2. Compilation. The compiler (*cc1*) translates the text file *hello.i* into the text file *hello.s*, which contains an assembly-language program.
3. Assembly. The assembler (*as*) translate *hello.s* into machine-language instructions packaged in a relocatable object file, object file for short, *hello.o*.
4. Linking. The linker (*ld*) takes *hello.o* and merges with any library functions that *hello.c* might call to create an executable. On Unix systems such executables do not have a suffix; on other systems it may have a *.exe* suffix.

C Programming: Recap

Pointer Declarations

```
char *pc;
unsigned int *pi;
long *pl;
```

Dereferencing

```
char c = *pc;
unsigned int i = *pi;
long l = *pl;
```

Pointer Arithmetic

You may perform arithmetic on pointers:

```
pc + 5
pi + 5
pl + 5
```

But you have to remember that the compiler takes the size of the underlying type into account. So if pc starts with value 2000, the expression pc + 5 returns 2005. However, if pi starts with value 2000, the expression pi + 5 returns 2020 (assuming sizeof(int) == 4), while if pl starts with value 2000, the expression pl + 5 returns 2040 (assuming sizeof(long) == 8).

Increment & Decrement Operators

- Unary operators (i.e., they take exactly 1 operand)
- ++ increments the operand by one
- −− decrements the operand by one
- Pre-increment: operator appears before the operand (e.g., ++i)
- Post-increment: operator appears after the operand (e.g., i++)
- Pre-decrement: operator appears before the operand (e.g., −−i)
- Post-decrement: operator appears after the operand (e.g., i−−)
- Pre-increment & pre-decrement: value of the expression based on the *incremented/decremented* operand value

- Post-increment & post-decrement: value of the expression based on the *original* operand value
- Increment and decrement operators on pointers: pointer arithmetic applies
- Contrast *++p (modifies p & returns the value at the incremented address) vs. *p++ (modifies p *but* returns the value at the original address)

Bit-Level Operations in C

One useful feature of C is that it supports bitwise Boolean operations, also know as bit vector operations.

There are six such operators:

- ~: applies the Boolean NOT operation on each bit of the operand (e.g., ~0x0F0F0F0Fu == 0xF0F0F0F0u)
- | : applies the Boolean OR operation on pairs of bits of the operands (e.g., 0x12345678u | 0x0FFF0000u == 0x1FFF5678u)
- & : applies the Boolean AND operation on pairs of bits of the operands (e.g., 0x12345678u | 0x0FFF0000u == 0x02340000u); warning: this is a case of an overloaded operator (i.e., an operator that means different things in different contexts), in some contexts, it can also mean "take the address of"
- ^ : applies the Boolean EXCLUSIVE-OR on pairs of bits of the operands (e.g., 0x12345678u ^ 0x0FFF0000u == 0x1DCB5678u)
- << : shifts the left operand by the number of bit positions indicated on the right (e.g., 0x12345678u << 4 == 0x23456780u); it is often the same as multiplying the left operand by 2^n, where *n* is the value of the right operand
- >> : shifts the left operand by the number of bit positions indicated on the right (e.g., 0x12345678u >> 4 == 0x01234567u); it is often the same as dividing (rounded down) the left operand by 2^n, where *n* is the value of the right operand; beware when applying this operator on signed types

You might ask what is the use of these operators. Some times programmers store information as bit vectors where some groups of bits together represent a value. Perhaps I want to store a floating point in a byte with the following format:

$se_3e_2e_1e_0f_2f_1f_0$

where *s* is the sign, *e* is the biased exponent, and *f* is the fraction.

Let's say I have variable *r* that holds such a floating point number and I want to know the value of *e*:

(*r* & 0x78) >> 3

Alternatively I want to set *e* to 6:

(*r* & 0x87) | ((0x6 & 0xF) << 3)

The C Preprocessor

Lines that start with the symbol "#" are preprocessor directives. The preprocessor is an early stage in the compiling process.

```
#include <headerfile>
```

or

```
#include "headerfile"
```

Include a header file; use angled brackets to include standard library include files (e.g., #include <stdio.h> or #include <limits.h>; use double quotes to include your own header files (e.g., #include "tinycalc.h").

```
#define IDENTIFIER VALUE
```

Will cause all occurrences of IDENTIFIER to be replaced by VALUE. Can be used, for instance, to define constants in C programs. The "const" keyword is a relatively recent addition to the C language, before that #define was the only way to define constants.

```
#define IDENTIFIER(ARG1, ARG2, ...) «TOKEN STRING THAT MAY INCLUDE ARG1, ARG2, ...»
```

An extension of the previous definition where the identifier takes arguments. Will cause all occurrences of IDENTIFIER(ARG1, ARG2, ...) to be replaced with a version of the «TOKEN STRING» that has actual arguments substituted for formal parameters.

Arithmetic and Logical Operations: Recap

The following table lists the x86-64 integer arithmetic operations.

Instruction	**Effect**	**Description**
leaq S, *D*	*D* ← &*S*	Load effective address
inc D	*D* ← *S* + 1	Increment
dec D	*D* ← *S* − 1	Decrement
neg D	*D* ← −*S*	Negate
not D	*D* ← ~*S*	Complement
add S, *D*	*D* ← *D* + *S*	Add
sub S, D	*D* ← *D* − *S*	Subtract
imul S, *D*	*D* ← *D* × *S*	Multiply
xor S, *D*	*D* ← *D* ^ *S*	Exclusive-or
or S, *D*	*D* ← *D* | *S*	Or
and S, *D*	*D* ← *D* & *S*	And
sal k, *D*	*D* ← *D* << *k*	Left shift
shl k, *D*	*D* ← *D* << *k*	Left shift (same as *sal*)
sar k, *D*	*D* ← *D* $>>_A$ *k*	Arithmetic right shift
shr k, *D*	*D* ← *D* $>>_L$ *k*	Logical right shift

Arithmetic and Logical Operations: Recap

The following table lists the x86-64 integer arithmetic operations.

Instruction	**Effect**	**Description**
leaq S, *D*	*D* ← &*S*	Load effective address
inc D	*D* ← *S* + 1	Increment
dec D	*D* ← *S* − 1	Decrement
neg D	*D* ← −*S*	Negate
not D	*D* ← ~*S*	Complement
add S, *D*	*D* ← *D* + *S*	Add
sub S, D	*D* ← *D* − *S*	Subtract
imul S, *D*	*D* ← *D* × *S*	Multiply
xor S, *D*	*D* ← *D* ^ *S*	Exclusive-or
or S, *D*	*D* ← *D* | *S*	Or
and S, *D*	*D* ← *D* & *S*	And
sal k, *D*	*D* ← *D* << *k*	Left shift
shl k, *D*	*D* ← *D* << *k*	Left shift (same as *sal*)
sar k, *D*	*D* ← *D* $>>_A$ *k*	Arithmetic right shift
shr k, *D*	*D* ← *D* $>>_L$ *k*	Logical right shift

Accessing Information: Recap

Registers

An x86-64 central processing unit (CPU) contains a set of 16 general-purpose registers storing 64-bit values.

Instructions can operate on data of different sizes stored in the low-order bytes of the 16 registers. Byte-level operations access the least significant byte, 16-bit operations access the least significant 2 bytes, and 32-bit operations access the least significant 4 bytes.

Different registers serve different roles in typical programs. Most unique among them is the stack pointer used to indicate the end position in the run-time stack. A set of standard programming conventions governs how the registers are to be used for managing the stack, passing function arguments, returning values from functions, and storing local (i.e., variables local to a function) and temporary data.

The following table lists these 16 registers, their names depending on the expected size of the data operands, and their roles in the standard programming convention used in this class.

Byte (8 bits)	Word (16 bits)	Long (32 bits)	Quad (64 bits)	Role
`%al`	`%ax`	`%eax`	`%rax`	Return value
`%bl`	`%bx`	`%ebx`	`%rbx`	Callee saved
`%cl`	`%cx`	`%ecx`	`%rcx`	4th argument
`%dl`	`%dx`	`%edx`	`%rdx`	3rd argument
`%sil`	`%si`	`%esi`	`%rsi`	2nd argument
`%dil`	`%di`	`%edi`	`%rdi`	1st argument
`%bpl`	`%bp`	`%ebp`	`%rbp`	Callee saved
`%spl`	`%sp`	`%esp`	`%rsp`	Stack pointer
`%r8b`	`%r8w`	`%r8d`	`%r8`	5th argument

%r9b	%r9w	%r9d	%r9	6th argument
%r10b	%r10w	%r10d	%r10	Caller saved
%r11b	%r11w	%r11d	%r11	Caller saved
%r12b	%r12w	%r12d	%r12	Callee saved
%r13b	%r13w	%r13d	%r13	Callee saved
%r14b	%r14w	%r14d	%r14	Callee saved
%r15b	%r15w	%r15d	%r15	Callee saved

Caller-saved registers are used to hold temporary values that need not be preserved across function calls. For this reason, it is the caller's responsibility to save the content of these registers (e.g., by pushing these registers onto the stack) if it needs said content after a function call.

Callee-saved registers are used to hold longer-lived values that should be preserved across function calls. When the caller makes a function call, it can expect that those registers will hold the same values after the function returns. For this reason, it is the callee's responsibility to save the content of these registers (e.g., by pushing these registers onto the stack) and to restore them before returning to the caller.

Operand Specifiers

Most instructions have one or more operands specifying the source values to use in performing an operation and the destination into which to place the result. x86-64 supports three operand types.

Type	**Form**	**Operand value**	**Name**
Immediate	$*Imm*	*Imm*	Immediate
Register	r_a	$R[r_a]$	Register
Memory	*Imm*(r_b, r_i, *s*)	M[*Imm* + $R[r_b]$ + $R[r_i]$ · *s*]	

With the Memory type, at least one of *Imm*, r_b, or r_i must be provided; if not specified the associated quantities (*Imm*, $R[r_b]$, $R[r_i]$ are assigned value 0). This gives rise to the following valid combinations and names.

Form	Operand value	Name

Imm	M[*Imm*]	Absolute
(r_b)	M[R[r_b]]	Indirect
Imm(r_b)	M[*Imm* + R[r_b]]	Base + displacement
(r_b, r_i)	M[R[r_b] + R[r_i]]	Indexed
Imm(r_b, r_i)	M[*Imm* + R[r_b] + R[r_i]]	Indexed
(, r_i, *s*)	M[R[r_i] · *s*]	Scaled indexed
Imm(, r_i, *s*)	M[*Imm* + R[r_i] · *s*]	Scaled indexed
(r_b, r_i, *s*)	M[R[r_b] + R[r_i] · *s*]	Scaled indexed
Imm(r_b, r_i, *s*)	M[*Imm* + R[r_b] + R[r_i] · *s*]	Scaled indexed

The scaling factor *s*, if specified, must be 1, 2, 4, or 8.

Data Movement Instructions

There are four simple move instructions: *movb*, *movw*, *movl*, and *movq*. They primarily differ in the amount of data that they transfer: 1, 2, 4, and 8 bytes, respectively. The source can be an Immediate, a Register, or Memory. The destination can be a Register or Memory. The sole restriction is that both cannot be Memory. When specifying a Register as a source or destination, its size designation (e.g., *%al*, *%ax*, *%eax*, or *%rax*) must match the type of the move instruction (i.e., match the suffix *b*, *w*, *l*, or *q* of the instruction). In most cases, the move instruction only updates the specific bytes of the register or memory destination. The only exception is when the destination is a 32-bit register; in that case the move instruction will also set the high-order 4 bytes of the register to 0.

The *movq* instruction can only have Immediate source operands that can be represented as 32-bit two's complement numbers. The value is then signed extended to produce the 64-bit value for the destination. The *movabsq* instruction can have an arbitrary Immediate value as its source operand and can only have a Register as a destination.

There are two groups of move instructions that deal with moving a value from a smaller size source to a larger size destination.

Instruction	Effect	Description
`movzbw`	$R_{16} \leftarrow \text{ZeroExtend}(S_8)$	Move zero-extended byte to word
`movzbl`	$R_{32} \leftarrow \text{ZeroExtend}(S_8)$	Move zero-extended byte to double word
`movzbq`	$R_{64} \leftarrow \text{ZeroExtend}(S_8)$	Move zero-extended byte to quad word
`movzwl`	$R_{32} \leftarrow \text{ZeroExtend}(S_{16})$	Move zero-extended word to double word

`movzwq`	$R_{64} \leftarrow \text{ZeroExtend}(S_{16})$	Move zero-extended double word to quad word
`movsbw`	$R_{16} \leftarrow \text{SignExtend}(S_8)$	Move sign-extended byte to word
`movsbl`	$R_{32} \leftarrow \text{SignExtend}(S_8)$	Move sign-extended byte to double word
`movsbq`	$R_{64} \leftarrow \text{SignExtend}(S_8)$	Move sign-extended byte to quad word
`movswl`	$R_{32} \leftarrow \text{SignExtend}(S_{16})$	Move sign-extended word to double word
`movswq`	$R_{64} \leftarrow \text{SignExtend}(S_{16})$	Move sign-extended word to quad word
`movslq`	$R_{64} \leftarrow \text{SignExtend}(S_{32})$	Move sign-extended double word to quad word
`cltq`	*%rax* ← SignExtend(*%eax*)	Move sign-extended %eax to %rax

The source operand must be either a Register or Memory; destination must be a Register. When specifying a Register as a source, its size designation must be appropriate for the instruction (i.e., must match the second to last suffix letter). Likewise the size designation of the destination Register must match the last suffix letter. In most cases the instruction performs the expected operation. The exception is when the destination is a 32-bit register; in that case will also set the high-order 4 bytes of the register to 0 (even in the sign-extending cases!).

The *cltq* instruction has no operands; it always uses *%eax* as its source and *%rax* as its destination. It performs the same operation as *movslq %eax, %rax* but it has a more compact encoding.

Pushing and Popping Stack Data

The final two data movement operations push data onto and pop data from the program stack.

Instruction	Effect	Description
`pushq S`	R[*%rsp*] ← R[*%rsp*] − 8 M[R[*%rsp*]] ← *S*	Push quad word
`popq D`	*D* ← M[R[*%rsp*]] R[*%rsp*] ← R[*%rsp*] + 8	Pop quad word

The source operand for the *pushq* instruction can be an Immediate, a Register, or Memory; while the destination operand for the *popq* instruction can be a Register or Memory.

Instruction	Effect	Description
`pushq` S	R[%rsp] $\leftarrow$ R[%rsp] $-$ 8; M[R[%rsp]] $\leftarrow$ S	Push quad word
`popq` D	D $\leftarrow$ M[R[%rsp]]; R[%rsp] $\leftarrow$ R[%rsp] $+$ 8	Pop quad word

Figure 3.8 **Push and pop instructions.**

Instruction		Effect	Description
`leaq`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D+1$	Increment
DEC	D	$D \leftarrow D-1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \,\&\, S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_{\mathrm{A}} k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_{\mathrm{L}} k$	Logical right shift

Figure 3.10 Integer arithmetic operations. The load effective address (`leaq`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation $>>_{\mathrm{A}}$ and $>>_{\mathrm{L}}$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

Instruction		Effect	Description
`imulq`	S	R[%rdx]:R[%rax] $\leftarrow$ $S \times$ R[%rax]	Signed full multiply
`mulq`	S	R[%rdx]:R[%rax] $\leftarrow$ $S \times$ R[%rax]	Unsigned full multiply
`cqto`		R[%rdx]:R[%rax] $\leftarrow$ SignExtend(R[%rax])	Convert to oct word
`idivq`	S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div S$	Signed divide
`divq`	S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div S$	Unsigned divide

Figure 3.12 Special arithmetic operations. These operations provide full 128-bit multiplication and division, for both signed and unsigned numbers. The pair of registers %rdx and %rax are viewed as forming a single 128-bit oct word.

`CF`: Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

`ZF`: Zero flag. The most recent operation yielded zero.

`SF`: Sign flag. The most recent operation yielded a negative value.

`OF`: Overflow flag. The most recent operation caused a two's-complement overflow—either negative or positive.

Instruction		Based on	Description
CMP	S_1, S_2	$S_2 - S_1$	Compare
`cmpb`			Compare byte
`cmpw`			Compare word
`cmpl`			Compare double word
`cmpq`			Compare quad word
TEST	S_1, S_2	S_1 & S_2	Test
`testb`			Test byte
`testw`			Test word
`testl`			Test double word
`testq`			Test quad word

Figure 3.13 Comparison and test instructions. These instructions set the condition codes without updating any other registers.

Instruction		Synonym	Effect	Set condition
`sete`	*D*	`setz`	*D* ← `ZF`	Equal / zero
`setne`	*D*	`setnz`	*D* ← `~ZF`	Not equal / not zero
`sets`	*D*		*D* ← `SF`	Negative
`setns`	*D*		*D* ← `~SF`	Nonnegative
`setg`	*D*	`setnle`	*D* ← `~(SF ^ OF) & ~ZF`	Greater (signed >)
`setge`	*D*	`setnl`	*D* ← `~(SF ^ OF)`	Greater or equal (signed >=)
`setl`	*D*	`setnge`	*D* ← `SF ^ OF`	Less (signed <)
`setle`	*D*	`setng`	*D* ← `(SF ^ OF) | ZF`	Less or equal (signed <=)
`seta`	*D*	`setnbe`	*D* ← `~CF & ~ZF`	Above (unsigned >)
`setae`	*D*	`setnb`	*D* ← `~CF`	Above or equal (unsigned >=)
`setb`	*D*	`setnae`	*D* ← `CF`	Below (unsigned <)
`setbe`	*D*	`setna`	*D* ← `CF | ZF`	Below or equal (unsigned <=)

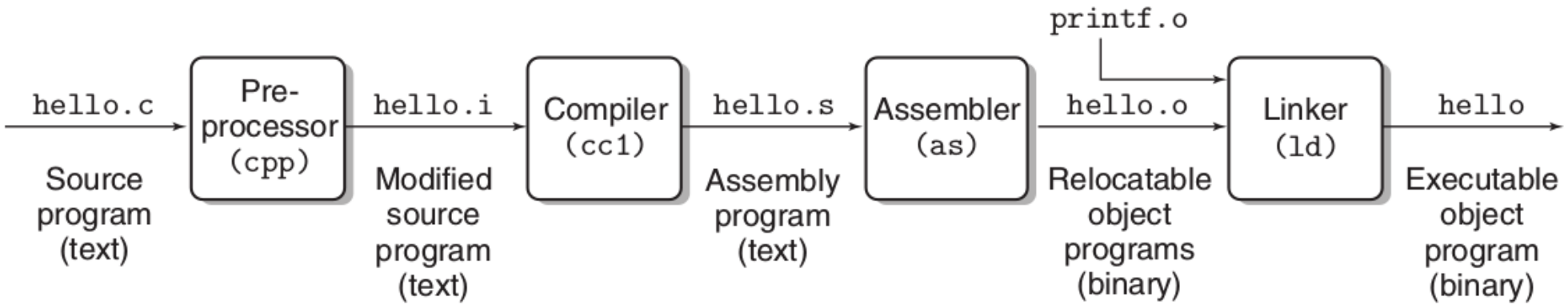
Figure 3.14 The SET instructions. Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," that is, alternate names for the same machine instruction.

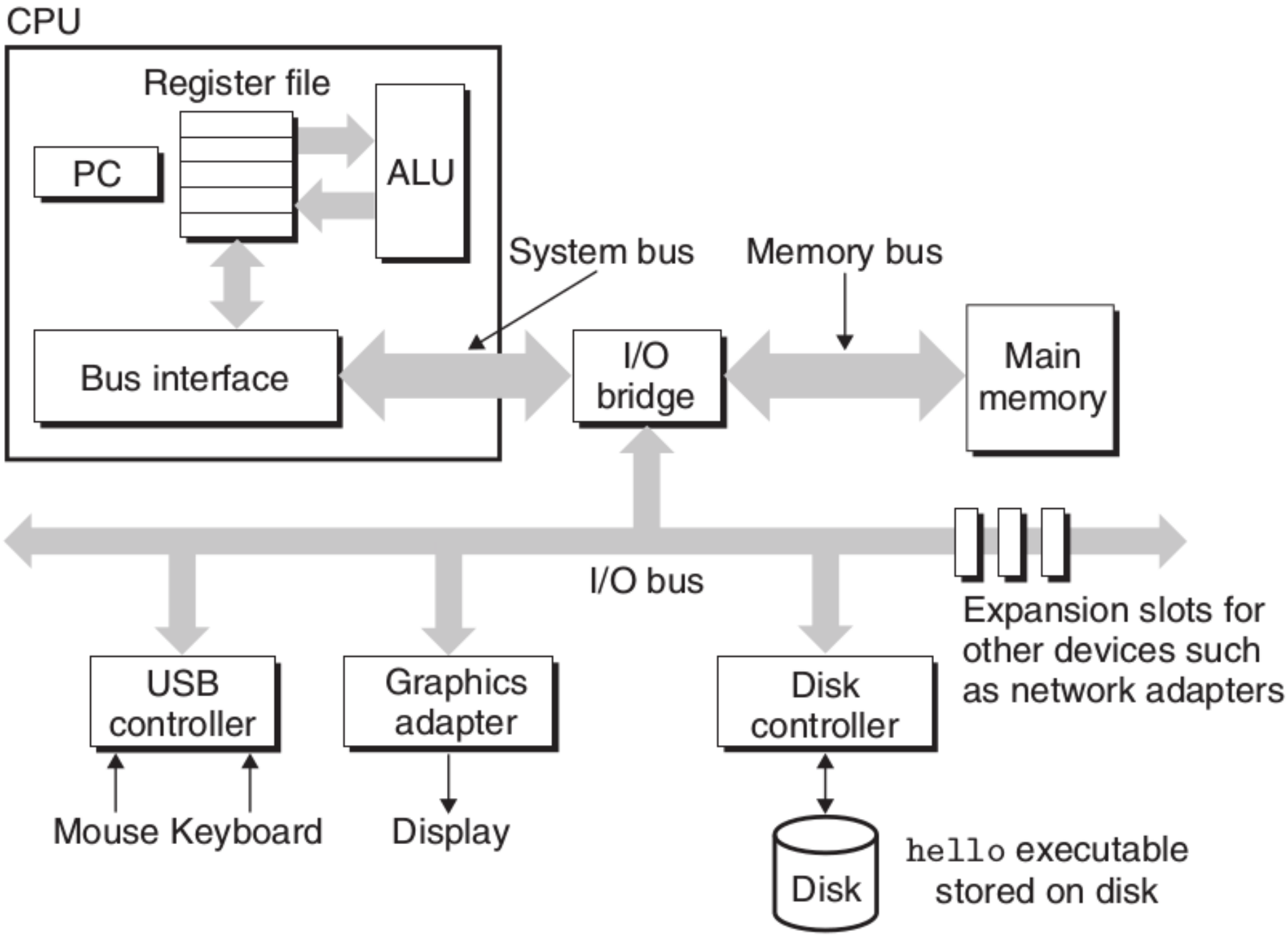
Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (unsigned <)
`jbe`	*Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

Figure 3.15 The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

Instruction		Synonym	Move condition	Description
`cmove`	S, R	`cmovz`	`ZF`	Equal / zero
`cmovne`	S, R	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	S, R		`SF`	Negative
`cmovns`	S, R		`~SF`	Nonnegative
`cmovg`	S, R	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	S, R	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	S, R	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	S, R	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	S, R	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	S, R	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	S, R	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	S, R	`cmovna`	`CF | ZF`	Below or equal (unsigned <=)

Figure 3.18 The conditional move instructions. These instructions copy the source value S to its destination R when the move condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.


hello.c
Pre-
processor
(cpp)
hello.i
Compiler
(cc1)
hello.s
Assembler
(as)
printf.o
hello.o
Linker
(ld)
hello
Source
program
(text)
Modified
source
program
(text)
Assembly
program
(text)
Relocatable
object
programs
(binary)
Executable
object
program
(binary)


CPU
Register file
PC
ALU
System bus
Memory bus
Bus interface
I/O bridge
Main memory
I/O bus
Expansion slots for other devices such as network adapters
USB controller
Graphics adapter
Disk controller
Mouse Keyboard
Display
Disk
hello executable stored on disk

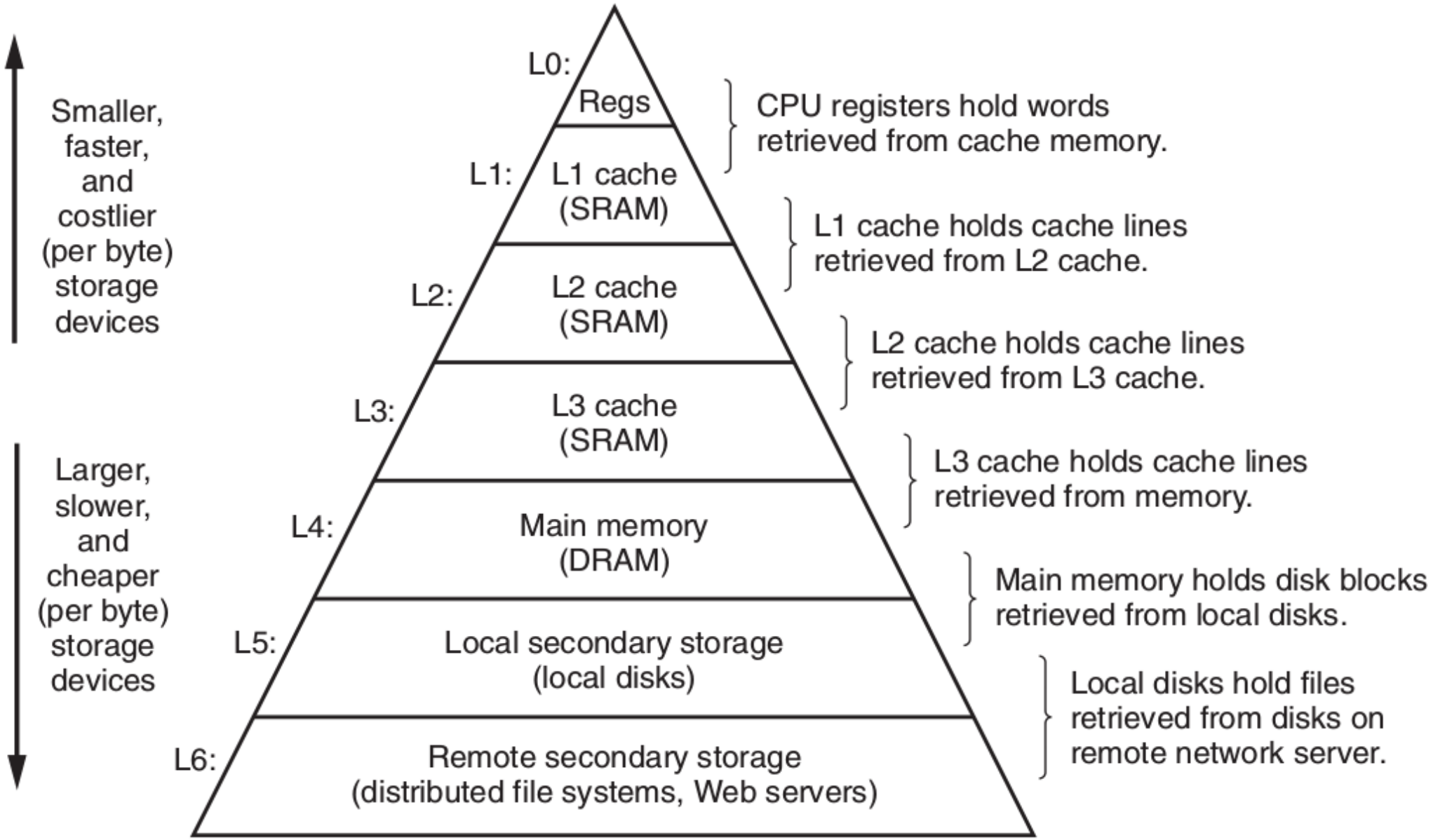


Figure 1.9 **An example of a memory hierarchy.**

C data type	Minimum	Maximum
[signed] char	−128	127
unsigned char	0	255
short	−32,768	32,767
unsigned short	0	65,535
int	−2,147,483,648	2,147,483,647
unsigned	0	4,294,967,295
long	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long	0	18,446,744,073,709,551,615
int32_t	−2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	0	18,446,744,073,709,551,615

Figure 2.10 **Typical ranges for C integral data types for 64-bit programs.**

Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\mathtt{r}_a$	$\mathsf{R}[\mathtt{r}_a]$	Register
Memory	Imm	$\mathsf{M}[Imm]$	Absolute
Memory	$(\mathtt{r}_a)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_a]]$	Indirect
Memory	$Imm(\mathtt{r}_b)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b]]$	Base + displacement
Memory	$(\mathtt{r}_b,\mathtt{r}_i)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i]]$	Indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i]]$	Indexed
Memory	$(,\mathtt{r}_i,s)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(,\mathtt{r}_i,s)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$(\mathtt{r}_b,\mathtt{r}_i,s)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i,s)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed

Figure 3.3 **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
`movb`			Move byte
`movw`			Move word
`movl`			Move double word
`movq`			Move quad word
`movabsq`	I, R	$R \leftarrow I$	Move absolute quad word

Figure 3.4 **Simple data movement instructions.**

Instruction	Effect	Description
MOVZ S, R	$R \leftarrow$ ZeroExtend(S)	Move with zero extension
`movzbw`		Move zero-extended byte to word
`movzbl`		Move zero-extended byte to double word
`movzwl`		Move zero-extended word to double word
`movzbq`		Move zero-extended byte to quad word
`movzwq`		Move zero-extended word to quad word

Figure 3.5 Zero-extending data movement instructions. These instructions have a register or memory location as the source and a register as the destination.

Instruction	Effect	Description
MOVS S, R	$R \leftarrow$ SignExtend(S)	Move with sign extension
`movsbw`		Move sign-extended byte to word
`movsbl`		Move sign-extended byte to double word
`movswl`		Move sign-extended word to double word
`movsbq`		Move sign-extended byte to quad word
`movswq`		Move sign-extended word to quad word
`movslq`		Move sign-extended double word to quad word
`cltq`	%rax $\leftarrow$ SignExtend(%eax)	Sign-extend %eax to %rax

Figure 3.6 Sign-extending data movement instructions. The MOVS instructions have a register or memory location as the source and a register as the destination. The `cltq` instruction is specific to registers %eax and %rax.