15-213 Handout #2: Machine-Level Programs on Linux/IA32*

Randal E. Bryant
David R. O'Hallaron

October 15, 1999

1 Introduction

This document describes how machine-level programs are encoded for Intel IA32 hardware running the LINUX operating system. The instruction set has a long, evolutionary development, as indicated by the following time line:

8086: (1978, 29K transistors). A 16-bit processor that formed the heart of the original IBM personal computers and MS-DOS. These machines were limited to a 640KB address space—addresses were only 20 bits long (1024KB addressable), and the operating system reserved 384KB of this for its own use.

80286: (1982, 134K transistors). Added more (and now obsolete) addressing modes. Formed the basis of the IBM PC-AT personal computer, the original platform for MS Windows.

i386: (1985, 275K transistors). Expanded the architecture to 32 bits. Added the "flat" addressing model that we will use. This was the first machine that could support the UNIX operating system.

i486: (1989, 1.9M transistors). Improved performance and integrated the floating-point unit onto the processor chip, but did not change the instruction set.

Pentium: (1993, 3.1M transistors). Improved performance, but only minor extensions of instruction set.

*Copyright © 1999, R. E. Bryant, D. R. O'Hallaron. All rights reserved.

Pentium/MMX: (1997, 4.5M transistors). Added new class of instructions for manipulating vectors of integers. Each datum can be 1, 2, or 4 bytes long. Each vector totals 64 bits.

Pentium II: (1997, 7M transistors). Introduced a radically new processor design. Only additions to instruction set was to add a class of "conditional move" instructions.

Pentium III: (1999, 8.2M transistors). Introduced yet another class of instructions for manipulating vectors of integer or floating point data. Each datum can be 1, 2, or 4 bytes, packed into vectors of 128 bits.

Each successive processor is designed to be backward compatible—able to run code compiled for any earlier version. As we will see, there are many strange artifacts in the instruction set due to this evolutionary heritage. Intel now calls its instruction set *IA32*, for "Intel Architecture 32-bit." You'll also hear the colloquial name "x86," reflecting the processor naming conventions up through the i486. They discontinued this naming convention when they lost a trademark infringement suit against their long-time rival AMD. The Trademark Office ruled that numbers can't be trademarked.

We describe the programs in assembly language, a low-level programming notation describing the exact machine-level instructions. This notation can be generated automatically with a C compiler. Suppose we have a C file `test.c` and run the command:

```
gcc -O2 -S test.c
```

This will generate a file `test.s` containing an assembly code version of the code the compiler generates. The command `gcc` indicates the Gnu C compiler. Since this is the default compiler on LINUX, one could also invoke it as simply `cc`. The flag `-O2` instructs the compiler to apply level-two optimizations. In general, increasing the level of optimization makes the final program run faster, but at a risk of increased compilation time and difficulties running debugging tools on your code. Level two optimization is a good compromise between optimized performance and ease of use. The `-S` flag indicates the compiler should generate assembly code rather than machine-level code.

The best references on IA32 are from Intel. Two useful references are [Intel-Arch] giving an overview of the architecture from the perspective of an assembly-language programmer, and [Intel-ISR] giving detailed descriptions of the different instructions. These references contain far more information than is required to understand Linux code. In particular, with flat mode addressing, all of the complexities of the segmented addressing scheme can be ignored. Appendix D of the Hennessy and Patterson computer architecture text [HP96] also describes IA32. Again, much of the discussion is on features that are irrelevant when using flat addressing.

The LINUX assembler uses the GAS (for "Gnu ASsembler") format. Unfortunately, there is very little documentation for this format specific to IA32. The Intel references use Intel's own assembly language syntax. Perhaps the most fundamental difference is that the operands are listed in the reverse order for the two notations! On a LINUX machine, running the command `info as`

C declaration	Intel Data Type	GAS suffix	Size (Bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double Word	`l`	4
`unsigned`	Double Word	`l`	4
`long int`	Double Word	`l`	4
`unsigned long`	Double Word	`l`	4
`char *`	Double Word	`l`	4
`float`	Double Word	`s`	4
`double`	Quad Word	`l`	8

Table 1: Sizes of standard data types

will display information about the assembler. One of the subsections documents machine-specific information, including a comparison of GAS with the more standard Intel notation. Note that GCC refers to these machines as "`i386`"—it generates code that could even run on a 14-year old i386. Neither GCC nor GAS support the more recent instruction set extensions, such as MMX and the conditional move instructions.

In this document, we only describe a subset of the IA32 architecture. IA32 is classified as a "CISC" (for Complex Instruction Set Computer) machine, and this is indeed an appropriate label. With its origin in the 1970's, and its continued accrual of new features, the machine has hundreds of different instructions, multiple addressing systems, and a number of data types. We present only the features encountered when executing typical integer code compiled using GCC.

2 Data Formats

Due to its origins as a 16-bit architecture that expanded into a 32-bit one, Intel uses the term "word" to refer to a 16-bit data type. Based on this, they refer to 32-bit quantities as "double words." They refer to 64-bit quantities as "quad words." We are mostly interested in bytes and double words.

Table 1 shows the machine representations used for the primitive data types of C. Note that most of the common data types are stored as double words. This includes both regular and long `int`'s, whether or not they are signed. In addition, all pointers (shown here as `char *`) are stored as 4-byte double words. Bytes are commonly used when manipulating string data.

As the table indicates, every operation in GAS has a single character suffix denoting the size of the operand. For example the `mov` (move data) instruction has 3 variants: `movb` (move byte), `movw` (move word), and `movel` (move double word). The suffix '`l`' is used for double words since on many machines 32-bit quantities are referred to as "long words." Note that GAS uses the suffix '`l`' to denote both a 4-byte integer as well as an 8-byte double precision floating point. This causes no

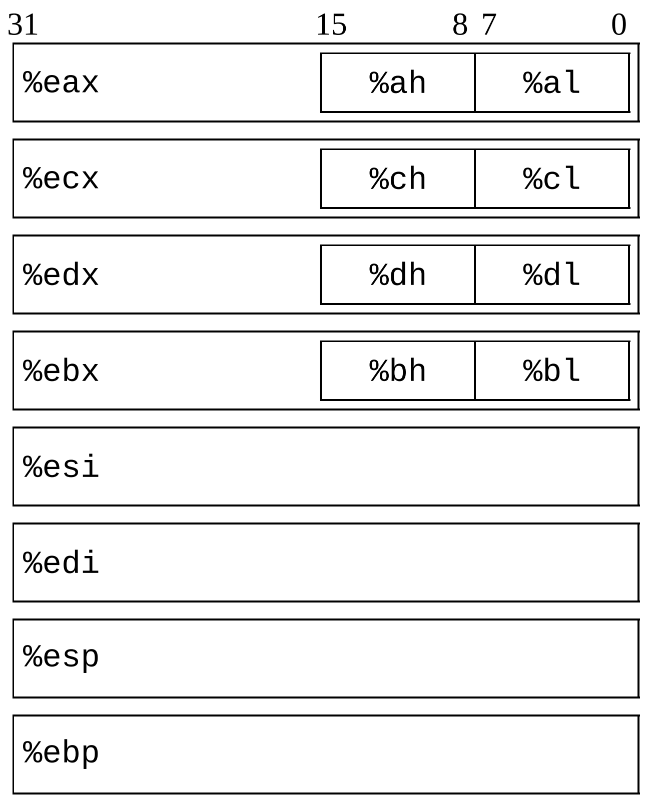

Figure 1: **Integer Registers.** All 8 registers can be accessed as 32-bits (double word). The low-order bytes of the first four registers can be accessed independently.

ambiguity, since floating point involves an entirely different set of instructions and registers.

3 Accessing Information

An IA32 central processing unit (CPU) contains a set of eight *registers* storing 32-bit values. These registers are used to store integer data as well as pointers. Figure 1 diagrams the eight registers. Their names all begin with `%e`, but otherwise have peculiar names. With the original 8086, the registers were 16-bits and each had a specific purpose. The names were chosen to reflect these different purposes. With flat addressing, the need for specialized registers is greatly reduced. For the most part, the first 6 registers can be considered general purpose registers with no restrictions placed on their use. The final two: `%ebp` and `%esp`, on the other hand, contain pointers to important places in the program stack. They should only be altered according to a set of standard conventions for stack management. It should be noted that some of the less common instructions do make use of specific registers. In addition, within procedures there are different

Type	Form	Operand Value	Name
Immediate	`$`Imm	Imm	Immediate
Register	$\mathtt{E}_a$	$Reg[\mathtt{E}_a]$	Register
Memory	Imm	$Mem[Imm]$	Absolute
Memory	$(\mathtt{E}_a)$	$Mem[Reg[\mathtt{E}_a]]$	Indirect
Memory	$Imm(\mathtt{E}_b)$	$Mem[Imm + Reg[\mathtt{E}_b]]$	Base + Displacement
Memory	$(\mathtt{E}_b,\mathtt{E}_i)$	$Mem[Reg[\mathtt{E}_b] + Reg[\mathtt{E}_i]]$	Indexed
Memory	$Imm(\mathtt{E}_b,\mathtt{E}_i)$	$Mem[Imm + Reg[\mathtt{E}_b] + Reg[\mathtt{E}_i]]$	Indexed
Memory	$(,\mathtt{E}_i,S)$	$Mem[Reg[\mathtt{E}_i] \cdot S]$	Scaled Indexed
Memory	$Imm(,\mathtt{E}_i,S)$	$Mem[Imm + Reg[\mathtt{E}_i] \cdot S]$	Scaled Indexed
Memory	$(\mathtt{E}_b,\mathtt{E}_i,S)$	$Mem[Reg[\mathtt{E}_b] + Reg[\mathtt{E}_i] \cdot S]$	Scaled Indexed
Memory	$Imm(\mathtt{E}_b,\mathtt{E}_i,S)$	$Mem[Imm + Reg[\mathtt{E}_b] + Reg[\mathtt{E}_i] \cdot S]$	Scaled Indexed

Table 2: **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.

conventions for saving and restoring the first three registers: `%eax`, `%ecx`, and `%edx`, than for the next three: `%ebx`, `%edi`, and `%esi`. This will be discussed in Section 6.

As indicated in Figure 1, the low-order two bytes of the first four registers can be independently read or written by the byte operation instructions. This feature was provided in the 8086 to allow backward compatibility to the 8008 and 8080—two 8-bit microprocessors that date back to 1974. When a byte instruction updates one of these single-byte "register elements," the remaining three bytes of the register do not change.

Most instructions have one or more *operands*, specifying the source values to reference in performing an operation and the destination location into which to place the result. As is typical of CISC instruction sets, IA32 supports a number of operand forms as indicated in Table 2. As shown, the different operand possibilities can be classified into three types. The first type, *immediate*, is for constant values. With GAS, these are written with a '`$`' followed by an integer using standard C notation, e.g., `$-577`, or `$0x1F`. The second type, *register*, denotes the contents of one of the registers, either one of the eight 32-bit registers (e.g., `%eax`) for a double word operation, or one of the eight single-byte register elements (e.g., `%al`) for a byte operation. In our table, we use the notation $\mathtt{E}_a$ to denote an arbitrary register a, and indicate its value with the reference $Reg[\mathtt{E}_a]$, viewing the set of registers as an array Reg indexed by register identifiers.

The third type of operand is a *memory* reference, in which we access some memory location according to a computed address, often called the *effective address*. As the table shows, there are many different forms for memory references. The most general form is shown at the bottom of the table with syntax $Imm(\mathtt{E}_b,\mathtt{E}_i,S)$. Such a reference has four components: an immediate offset Imm, a base register $\mathtt{E}_b$, an index register $\mathtt{E}_i$, and a scale factor S, where S must be 1, 2, 4, or 8. The effective address is then computed as $Imm + Reg[\mathtt{E}_b] + Reg[\mathtt{E}_i] \cdot S$. This general form is often

Instruction		Effect	Description
`movl`	S, D	$D \leftarrow S$	Move Double Word
`movb`	S, D	$D \leftarrow S$	Move Byte
`movsbl`	S, D	$D \leftarrow \mathit{SignExtend}(S)$	Move Sign-Extended Byte
`pushl`	S	$\mathit{Reg}[\texttt{\%esp}] \leftarrow \mathit{Reg}[\texttt{\%esp}] - 4$; $\mathit{Mem}[\mathit{Reg}[\texttt{\%esp}]] \leftarrow S$	Push
`popl`	D	$D \leftarrow \mathit{Mem}[\mathit{Reg}[\texttt{\%esp}]]$; $\mathit{Reg}[\texttt{\%esp}] \leftarrow \mathit{Reg}[\texttt{\%esp}] + 4$	Pop

Table 3: **Data Movement Instructions.**

seen when referencing elements of arrays. The other forms are simply special cases of this general form where some of the components are omitted.

Among the most heavily used instructions are ones that perform data movement. The generality of the operand notation allows a simple move instruction to perform what in many machines would require a number of instructions. Table 3 lists the important data movement instructions. The most common is the `movl` instruction for moving double words. The source operand designates a value that is either immediate, stored in a register, or stored in memory. The destination operand designates a location that is either a register or a memory address. IA32 imposes the restriction that `movl` cannot have both operands refer to memory locations. Copying a value from one memory location to another requires two instructions—the first to load the source value into a register, and the second to write this register value to the destination.

The following are some examples of `movl` instructions showing the five possible combinations of source and destination types:

```
movl $0x4050, %eax        Immediate–Register
movl %ebp, %esp           Register–Register
movl (%edi,%ecx), %eax    Memory–Register
movl $-17, (%esp)         Immediate–Memory
movl %eax,-12(%ebp)       Register–Memory
```

The `movb` instruction is similar, except that it moves a single byte. When one of the operands is a register, it must be one of the eight single-byte register elements illustrated in Figure 1.

The `movsbl` instruction takes a single-byte source operand, performs a sign extension to 32 bits (i.e., it sets the high order 24 bits to the most significant bit of the source byte), and copies this to a double-word destination. Thus the instructions `movb %dh, %al` and `movsbl %dh, %eax` have very different effects. Both of them set the low order byte of register `%eax` to the second byte of `%edx`. The former does not change the other three bytes, while the latter sets these bits to either all 0's or all 1's, according to the most significant bit of the copied byte.

The final two data movement operations are used to push data onto and pop data from the program

Instruction		Effect	Description
`leal`	S, D	$D \leftarrow$ `&`S	Load Effective Address
`incl`	D	$D \leftarrow D$ `+ 1`	Increment
`decl`	D	$D \leftarrow D$ `- 1`	Decrement
`negl`	D	$D \leftarrow$ `-`D	Negate
`notl`	D	$D \leftarrow$ `~`D	Complement
`addl`	S, D	$D \leftarrow D$ `+` S	Add
`subl`	S, D	$D \leftarrow D$ `-` S	Subtract
`imull`	S, D	$D \leftarrow D$ `*` S	Multiply
`xorl`	S, D	$D \leftarrow D$ `^` S	Exclusive-Or
`orl`	S, D	$D \leftarrow D$ `|` S	Or
`andl`	S, D	$D \leftarrow D$ `&` S	And
`sall`	k, D	$D \leftarrow D$ `<<` k	Left Shift
`shll`	k, D	$D \leftarrow D$ `<<` k	Left Shift
`sarl`	k, D	$D \leftarrow D$ `>>` k	Arithmetic Right Shift
`shrl`	k, D	$D \leftarrow D$ `>>` k	Logical Right Shift

Table 4: **Integer Arithmetic Operations.** The Load Effective Address `leal` is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. Note the nonstandard ordering of the operands with GAS.

stack. Both the `pushl` and the `popl` instructions take a single operand—the data source for pushing and the data destination for popping. The program stack is stored in some region of memory. The stack grows downward such that the top element of the stack has the lowest address of all stack elements. The stack pointer `%esp` holds the address of this lowest stack element. Pushing a double word value onto the stack therefore involves first decrementing the stack pointer by 4 and then writing the value at the new top of stack address. Popping a double word involves reading from the top of stack location and then incrementing the stack pointer by 4.

4 Arithmetic and Logical Operations

Table 4 lists some of the double word integer operations, divided into four groups. Note that all of them have two operands. These operands are specified using the same notation as described in Section 3. With the exception of `leal`, each of these instructions has a counterpart that operates on bytes. The suffix ‘`l`’ is replaced by ‘`b`’ for the byte operation. For example, `addl` becomes `addb`.

4.1 Load Effective Address

The Load Effective Address `leal` instruction is actually a variant of the `movl` instruction. That is, its first operand appears to be a memory reference, but instead of reading from the designated location the instruction copies the effective address to the destination. We indicate this computation using the C address operator &S. This instruction can be used to generate pointers for later memory references. In addition, it can be used to compactly describe common arithmetic operations. For example, if register `%edx` contains value x, then the instruction `leal 7(%edx,%edx,4), %eax` will set register `%eax` to $5x + 7$. The destination operand must be a register.

4.2 Unary and Binary Operations

The second group are unary operations, with the single operand serving as both source and destination. This operand can be either a register or a memory location. Thus, the instruction `incl (%esp)` causes the element on the top of the stack to be incremented.

The third group are binary operations, where the second operand is used as both a source and a destination. This syntax is reminiscent of the C assignment operators such as `+=`. Observe, however, that the source operand is given first and the destination second. This looks peculiar for noncommutative operations. For example, the instruction `subl %eax,%edx` decrements register `%edx` by the value in `%eax`. The first operand can be either an immediate value, a register, or a memory location. The second can be either a register or a memory location. As with the `movl` instruction, however, the two operands cannot both be memory locations.

4.3 Shift Operations

The final group are shift operations where the shift amount is given first, and the value to shift is given second. Both arithmetic and logical right shifts are possible. The shift amount is encoded as a single byte, since only shifts of between 0 and 31 are allowed. The shift amount is given either as an immediate or in single-byte register element `%cl`. As Table 4 indicates, there are two names for the left shift instruction: `sall` and `shll`. Both have the same effect, filling from the right with 0's. The right shift instructions differ in that `sarl` performs an arithmetic shift (fill with copies of the sign bit), whereas `shrl` performs a logical shift (fill with 0's).

Note that, with the exception of the right shift operations, none of the instructions distinguish between signed and unsigned operands. Two's complement arithmetic has the same bit-level behavior as unsigned arithmetic for all of the instructions listed.

Instruction	Effect	Description
`imull` S	Reg[`%edx`]:Reg[`%eax`] $\leftarrow$ $S \times Reg$[`%eax`]	Signed Full Multiply
`mull` S	Reg[`%edx`]:Reg[`%eax`] $\leftarrow$ $S \times Reg$[`%eax`]	Unsigned Full Multiply
`cltd`	Reg[`%edx`]:Reg[`%eax`] $\leftarrow$ $SignExtend(Reg$[`%eax`]$)$	Convert to Quad Word
`idivl` S	Reg[`%edx`] $\leftarrow$ Reg[`%edx`]:Reg[`%eax`] $\mathrm{mod}\ S$; Reg[`%eax`] $\leftarrow$ Reg[`%edx`]:Reg[`%eax`] $\div S$	Signed Divide
`divl` S	Reg[`%edx`] $\leftarrow$ Reg[`%edx`]:Reg[`%eax`] $\mathrm{mod}\ S$; Reg[`%eax`] $\leftarrow$ Reg[`%edx`]:Reg[`%eax`] $\div S$	Unsigned Divide

Table 5: **Special Arithmetic Operations.** These operations provide full 64-bit multiplication and division, for both signed and unsigned numbers. The pair of registers `%edx` and `%eax` are viewed as forming a single 64-bit quad word.

4.4 Special Arithmetic Operations

Table 5 describes instructions that support generating the full 64-bit product of two 32-bit numbers, as well as integer division.

The `imull` instruction listed in Table 4 is known as the “two-operand” multiply instruction. It generates a 32-bit product from two 32-bit operands, implementing the operations $*^{u}_{32}$ and $*^{t}_{32}$ described in the handout on integer arithmetic. Recall that when truncating the product to 32 bits, both unsigned multiply and two’s complement multiply have the same bit-level behavior. IA32 also provides two different “one-operand” multiply instructions to compute the full 64-bit product of two 32-bit values—one for unsigned (`mull`), and one for two’s complement (`imull`) multiplication. For both of these, one argument must be in register `%eax`, and the other is given as the instruction source operand. The product is then stored in registers `%edx` (high-order 32 bits) and `%eax` (low-order 32 bits.)

As an example, suppose we have signed numbers `x` and `y` stored in positions -8 and -12 relative to `%ebp`, and we want to store their full 64-bit product as 8 bytes on top of the stack. The code would proceed as follows:

```
movl -8(%ebp), %eax   Put x in %eax
imull -12(%ebp)       Multiply by y
pushl %edx            Push high-order 32-bits
pushl %eax            Push low-order 32-bits
```

Observe that the order in which we push the two registers is correct for a Little Endian machine in which the stack grows toward lower addresses, i.e., the low-order bytes of the product will have lower addresses than the high-order bytes.

Note that Table 4 does not list any division or modulus operations. These operations are provided by the single-operand divide instructions similar to the single-operand multiply instructions. The signed division instruction `idivl` takes as dividend the 64-bit quantity in registers `%edx` (high-order 32 bits) and `%eax` (low-order 32 bits). The divisor is given as the instruction operand. The instructions store the quotient in register `%eax` and the remainder in register `%edx`. The `cltd`[1] instruction can be used to form the 64-bit dividend from a 32-bit value stored in register `%eax`. This instruction sign extends `%eax` into `%edx`.

As an example, suppose we have signed numbers `x` and `y` stored in positions -8 and -12 relative to `%ebp`, and we want to store values `x / y` and `x % y` on the stack. The code would proceed as follows:

`movl -8(%ebp), %eax`	Put `x` in `%eax`
`cltd`	Sign extend into `%edx`
`idivl -12(%ebp)`	Divide by `y`
`pushl %eax`	Push `x / y`
`pushl %edx`	Push `x % y`

The `divl` instruction performs unsigned division. Usually register `%edx` is set to 0 beforehand.

5 Control

5.1 Condition Codes

In addition to the integer registers, the CPU maintains a set of single-bit *condition code* registers describing attributes of the most recent arithmetic or logical operation. These registers can then be tested to perform conditional branches. The most useful condition codes are:

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.

For example, suppose we used the `addl` instruction to perform the equivalent of the C expression `t = a+b`, where variables `a`, `b`, and `t` are of type `int`. Then the condition codes would be set according to the following C statements:

```
CF = (unsigned t) < (unsigned a); /* Unsigned overflow */
ZF = (t == 0);
```

[1]This instruction is called `cdq` in the Intel documentation, one of the few cases where the GAS name for an instruction bears no relation to the Intel name.

```
SF = (t < 0);
OF = (a < 0 == b < 0) && (t < 0 != a < 0); /* Signed overflow */
```

Note that the `leal` instruction does not alter any condition codes, since it is intended to be used in address computations. Otherwise, all of the instructions listed in Table 4 cause the condition codes to be set. For the logical operations, such as `xorl`, the carry and overflow flags are set to 0. For the shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to 0.

In addition to the operations of Table 4, two operations (having both 32-bit and 8-bit forms) set conditions codes without altering any other registers:

Instruction	Based on	Description
`cmpl` S_2, S_1	S_1 - S_2	Compare Double Word
`testl` S_2, S_1	S_1 & S_2	Test Double Word
`cmpb` S_2, S_1	S_1 - S_2	Compare Byte
`testb` S_2, S_1	S_1 & S_2	Test Byte

The `cmpl` and `cmpb` instructions set the condition codes according to the difference of their two operands. Note that the operands are listed in reverse order, making the code more difficult to read. These instructions set the zero flag if the two operands are equal. The other flags can be used to determine ordering relations between the two operands.

The `testl` and `testb` instructions set the zero and negative flags based on the AND of their two operands. Typically, the same operand is repeated twice, e.g., `testl %eax,%eax`, or one of the operands is a mask indicating which bits should be tested.

5.2 Accessing the Condition Codes

Rather than reading the condition codes directly, the two most common methods of accessing them are to set an integer register based on some combination of condition codes, or to perform a conditional branch based on some combination of condition codes.

The different `set` instructions described in Table 6 set a single byte to 0 or to 1 depending on some combination of the conditions codes. The destination operand is either one of the eight single-byte register elements or a memory location. To generate a 32-bit result, we must also clear the high order 24 bits. A typical instruction sequence for a C predicate such as `a < b` is therefore as follows:

```
 # a in %ecx, b in %edx
cmpl %edx,%ecx   # a:b
setl %al         # Set low order byte of %eax
andl $0xFF,%eax  # Clear remaining bytes of %eax
```

Instruction	Effect	Set Condition
`sete` D	$D \leftarrow$ `ZF`	Equal / Zero
`setne` D	$D \leftarrow$ `~ZF`	Not Equal / Not Zero
`sets` D	$D \leftarrow$ `SF`	Negative
`setns` D	$D \leftarrow$ `~SF`	Nonnegative
`setg` D	$D \leftarrow$ `~(SF ^ OF) & ~ZF`	Greater (Signed >)
`setge` D	$D \leftarrow$ `~(SF ^ OF)`	Greater or Equal (Signed >=)
`setl` D	$D \leftarrow$ `SF ^ OF`	Less (Signed <)
`setle` D	$D \leftarrow$ `(SF ^ OF) | ZF`	Less or Equal (Signed <=)
`seta` D	$D \leftarrow$ `~CF & ~ZF`	Above (Unsigned >)
`setae` D	$D \leftarrow$ `~CF`	Above or Equal (Unsigned >=)
`setb` D	$D \leftarrow$ `CF`	Below (Unsigned <)
`setbe` D	$D \leftarrow$ `CF & ~ZF`	Below or Equal (Unsigned <=)

Table 6: **The `set` Instructions.** Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes.

Although all arithmetic operations set the condition codes, the descriptions of the different `set` commands apply to the case where a comparison instruction has been executed, setting the condition codes according to the computation `t = a-b`. For example, consider the `sete`, or "Set when equal" instruction. When `a` $=$ `b`, we will have `t` $= 0$, and hence the zero flag indicates equality.

Similarly, consider testing a signed comparison with the `setl`, or "Set when less," instruction. When `a` and `b` are in two's complement form, then for `a` $<$ `b` we will have `a` $-$ `b` < 0 if the true difference were computed. When there is no overflow, this would be indicated by having the sign flag set. When there is positive overflow, however, we will have `t` < 0, and when there is negative overflow, we will have `t` > 0. In either case, the sign flag will indicate the opposite of the sign of the true difference. Hence, the EXCLUSIVE-OR of the overflow and sign bits provides a test for whether `a` $<$ `b`. The other signed comparison tests are based on other combinations of `SF ^ OF` and `ZF`.

For the testing of unsigned comparisons, the carry flag will be set by the `cmpl` instruction when the integer difference `a` $-$ `b` of the unsigned arguments `a` and `b` would be negative, i.e., when `(unsigned) a < (unsigned) b`. Thus, these tests use combinations of the carry and zero flags.

5.3 Jumping

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. These jump

Instruction	Jump Condition	Description
`jmp` *Label*	Always	Equal / Zero
`je` *Label*	`ZF`	Equal / Zero
`jne` *Label*	`~ZF`	Not Equal / Not Zero
`js` *Label*	`SF`	Negative
`jns` *Label*	`~SF`	Nonnegative
`jg` *Label*	`~(SF ^ OF) & ~ZF`	Greater (Signed >)
`jge` *Label*	`~(SF ^ OF)`	Greater or Equal (Signed >=)
`jl` *Label*	`SF ^ OF`	Less (Signed <)
`jle` *Label*	`(SF ^ OF) | ZF`	Less or Equal (Signed <=)
`ja` *Label*	`~CF & ~ZF`	Above (Unsigned >)
`jae` *Label*	`~CF`	Above or Equal (Unsigned >=)
`jb` *Label*	`CF`	Below (Unsigned <)
`jbe` *Label*	`CF & ~ZF`	Below or Equal (Unsigned <=)

Table 7: **The `jump` Instructions.** These instructions jump to a labeled destination when the jump condition holds.

destinations are generally indicated by a *label*. Consider the following assembly code sequence:

```
      xorl %eax,%eax   # Set %eax to 0
      jmp L1           # Goto L1
      movl (%eax),%edx # Null pointer dereference
L1:
      popl %edx
```

The instruction `jmp L1` will cause the program to skip over the `movl` instruction, and instead resume execution with the `popl` instruction. One job of the assembler is then to determine the actual addresses for the labeled instructions and fill in the fields of the jump instruction appropriately.

The `jmp` intruction jumps unconditionally. The other jump instructions either jump or continue executing at the next instruction in the code sequence depending on some combination of the condition codes. Note that the names of these instructions and the conditions under which they jump match those of the `set` instructions.

6 Procedures

IA32 programs make use of the program stack to support procedure calls. The stack is used to pass procedure arguments, to store return information, to save registers for later restoration, and for local storage. The portion of the stack allocated for a single procedure call is called a *stack*

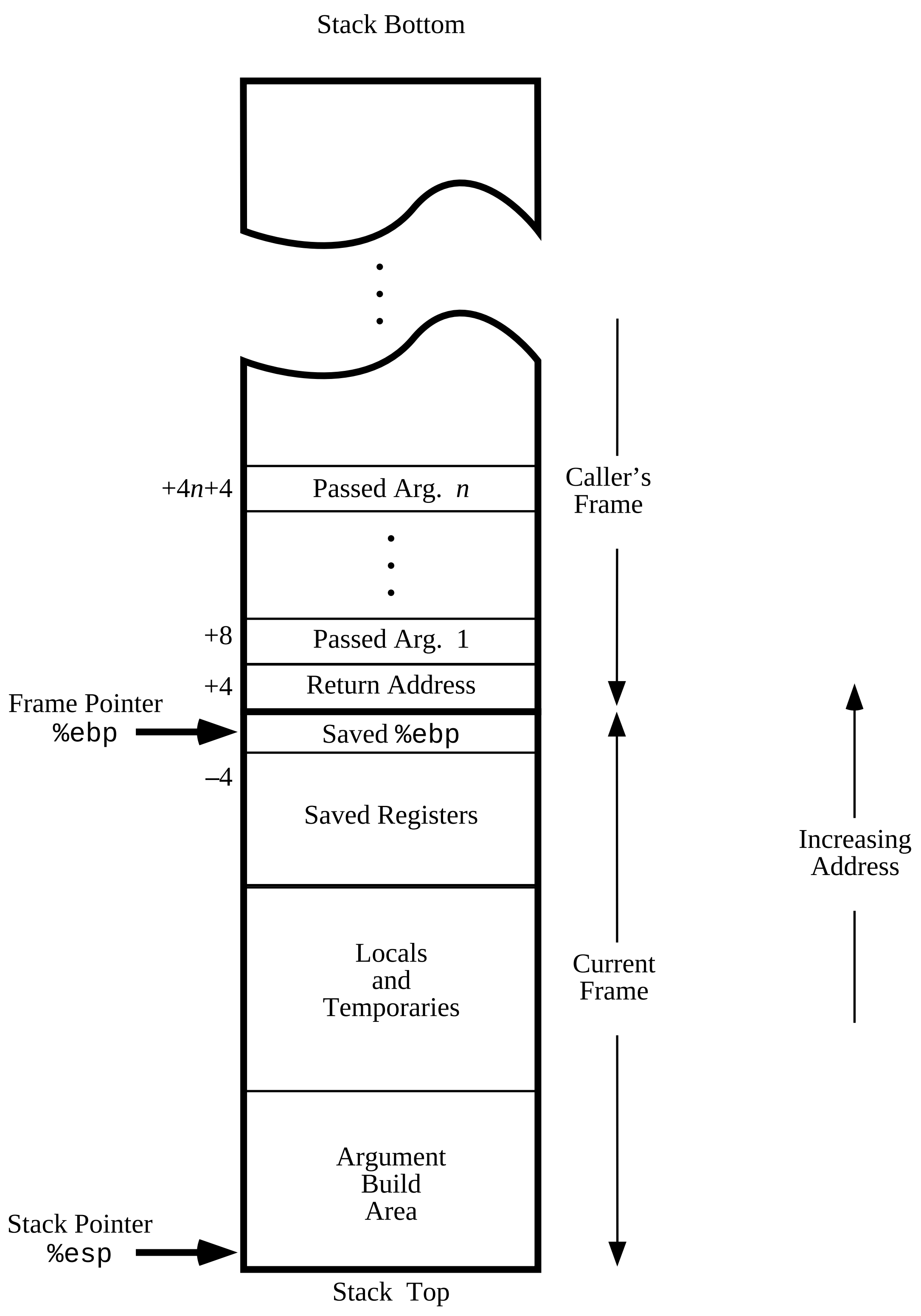

Figure 2: **Stack Frame Structure.** The stack is used for passing arguments, for storing return information, for saving registers, and for local storage.

frame. Figure 2 diagrams the general structure of a stack frame. The topmost stack frame is delimited by two pointers, with register `%ebp` serving as the *frame pointer*, and register `%esp` serving as the *stack pointer*. The stack pointer can move while the procedure is executing, and hence most information is accessed relative to the frame pointer.

Suppose procedure `P` (the *caller*) calls procedure `Q` (the *callee*.) The arguments to `Q` are contained within the stack frame for `P`. In addition, when `P` calls `Q`, the *return address* within `P` where the program should resume execution when it returns from `Q` will be pushed on the stack. This forms the end of `P`'s stack frame. The stack frame for `Q` begins with the saved value of the frame pointer (i.e., `%ebp`.) This is followed by copies of any other saved register values.

Procedure `Q` also uses the stack for any local variables that cannot be stored in registers. This can occur for the following reasons:

- There are not enough registers to hold all of the local data.
- Some of the local variables are arrays or structures and hence must be accessed by array or structure references.
- The address operator '`&`' is applied to one of the local variables, and hence we must be able to generate an address for it.

Finally, `Q` will use the stack frame for storing arguments to any procedures it calls.

By convention, registers `%eax`, `%edx`, and `%ecx` are classified as *caller save* registers. This means that `Q` can overwrite these registers without destroying any data required by `P`. On the other hand, registers `%ebx`, `%esi`, and `%edi` are classified as *callee save* registers. This means that `Q` must save the values of any of these registers on the stack before overwriting them, and restore them before returning, since `P` (or some higher-level procedure) may need these values for its future computations. In addition, registers `%ebp` and `%esp` must be maintained according to the conventions described here.

The instructions supporting procedure calls and returns are as follows:

Instruction	Description
`call` *Label*	Procedure Call
`leave`	Prepare stack for return
`ret`	Return from call

The `call` instruction has a label as its operand where the code for the called procedure starts. The effect of this instruction is to push a return address on the stack and jump to the start of the called procedure. The return address is the address of the instruction immediately following the `call` in the program, so that execution will resume at this location when the called procedure returns. The `ret` instruction pops an address of the stack and jumps to this location. The proper use of this instruction is to have prepared the stack so that the stack pointer points to the place where

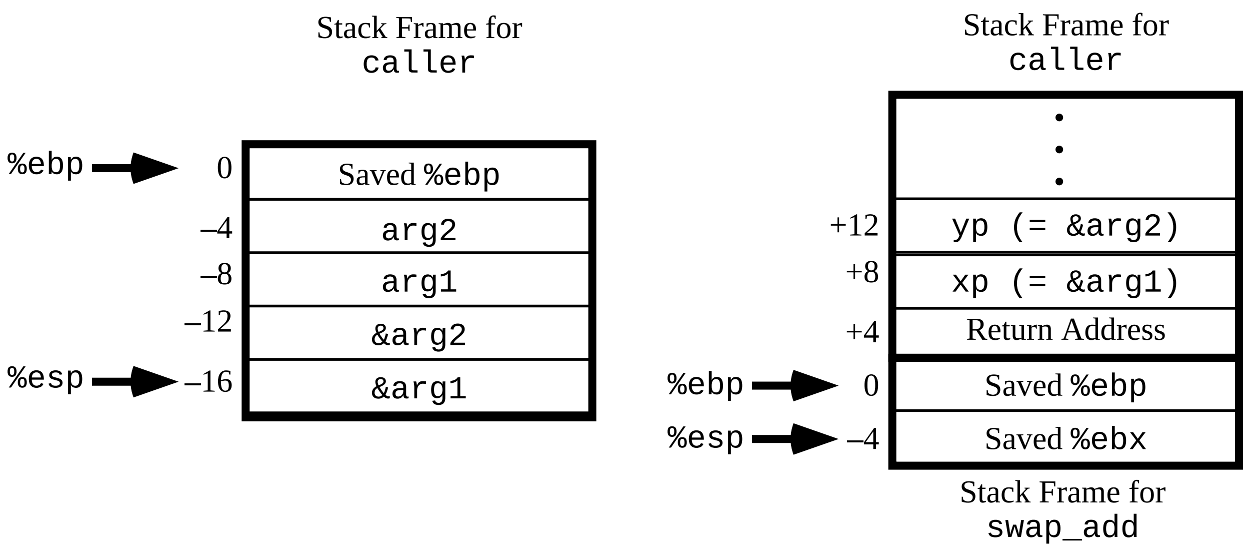

Figure 3: **Stack Frames for `caller` and `swap_add`.** Procedure `swap_add` retrieves its arguments from the stack frame for `caller`.

the preceding `call` instruction stored its return address. The `leave` instruction can be used to prepare the stack for returning. It is equivalent to the following code sequence:

`movl %ebp, %esp`	Set stack pointer to beginning of frame
`popl %ebp`	Restore old version of `%ebp` and set stack pointer to end of caller's frame

Alternatively, this preparation can be performed by an explicit sequence of move and pop operations.

Register `%eax` is used for returning the value of any function that returns an integer or pointer.

As an example, consider the following C procedures:

```
int caller()                           swap_add(int *xp, int *yp)
{                                      {
  int arg1 = 534;                        int x = *xp;
  int arg2 = 1057;                       int y = *yp;
  int sum = swap_add(&arg1, &arg2);      *xp = y;
  int diff = arg1 - arg2;                *yp = x;
  return sum * diff;                     return x+y;
}                                      }
```

Figure 3 shows the stack frames for the two procedures. Observe that `swap_add` retrieves its arguments from the stack frame for `caller`. These locations are accessed relative to the frame pointer in register `%ebp`. The numbers along the left of the frames indicate the address offsets relative to the frame pointer.

The stack frame for `caller` includes storage for local variables `arg1` and `arg2`, at positions -8 and -4 relative to the frame pointer. These variables must be stored on the stack, since we must generate addresses for them. The following assembly code from the compiled version of `caller` shows how it calls `swap_add`.

```
C1  leal -4(%ebp),%eax  Compute &arg2
C2  pushl %eax          Push &arg2
C3  leal -8(%ebp),%eax  Compute &arg1
C4  pushl %eax          Push &arg1
C5  call _swap_add      Call swap_add
```

Observe that this code computes the addresses of local variables `arg2` and `arg1` (using the `leal` instruction) and pushes them on the stack. It then calls `swap_add`.

The compiled code for `swap_add` has three parts: the "setup" where the stack frame is initialized, the "body" where the actual computation of the procedure is performed, and the "finish" where the stack state is restored and the procedure returns.

The following is the setup code for `swap_add`. Recall that the `call` instruction will already push the return address on the stack.

```
S1  pushl %ebp        Save old %ebp
S2  movl %esp,%ebp    Set %ebp
S3  pushl %ebx        Save %ebx
```

Procedure `swap_add` requires register `%ebx` for temporary storage. Since this is a callee-save register, it pushes the old value on the stack as part of the stack frame setup.

The following is the body code for `swap_add`:

```
B1  movl 8(%ebp),%edx   Get xp
B2  movl 12(%ebp),%ecx  Get yp
B3  movl (%edx),%ebx    Get x
B4  movl (%ecx),%eax    Get y
B5  movl %eax,(%edx)    Store y at *xp
B6  movl %ebx,(%ecx)    Store x at *yp
B7  addl %ebx,%eax      Return value = x+y
```

This code retrieves its arguments from the stack frame for `caller`. Since the frame pointer has shifted, the locations of these arguments has shifted from positions -12 and -16 relative to the old value of `%ebp` to positions $+12$ and $+8$ relative to new value of `%ebp`. Observe that the sum of variables `x` and `y` is stored in register `%eax` to be passed as the returned value.

The following is the finishing code for `swap_add`:

```
F1  movl -4(%ebp),%ebx  Restore %ebx
F2  movl %ebp,%esp      Restore %esp
F3  popl %ebp           Restore %ebp
F4  ret                 Return
```

This code simply restores the values of the 3 registers `%ebx`, `%esp`, and `%ebp`, and then executes the `ret` instruction. Note that instructions F2 and F3 could be replaced by a single `leave` instruction. Different versions of GCC seem to have different preferences in this regard.

The following code in `caller` comes immediately after the instruction calling `swap_add`:

```
R1  movl %eax,%edx   Resume here
```

Upon return from `swap_add`, procedure `caller` will resume execution with this instruction. Observe that this instruction copies the return value from `%eax` to a different register.

References

[HP96] J. L. Hennessy, and D. A. Patterson, "An Alternative to RISC: The Intel 80x86," Appendix A of: *Computer Architecture: A Quantitative Approach*, 2nd edition Morgan-Kaufmann, San Francisco, 1996.

[Intel-Arch] Intel Corporation, *Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, Order Number 243190, Intel Corporation, 1999. Adobe Acrobat version available in file:
`/afs/cs.cmu.edu/academic/class/15213-f99/doc/intel-arch.pdf`

[Intel-ISR] Intel Corporation, *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, Order Number 243191, Intel Corporation, 1999. Adobe Acrobat version available in file:
`/afs/cs.cmu.edu/academic/class/15213-f99/doc/intel-isr.pdf`