The x86 Instruction Set Architecture[1] CS232: Computer Architecture II

This set of notes provides an overview of the x86 instruction set architecture and its use in modern software. The goal is to familiarize you with the ISA to the point that you can code simple programs and can read disassembled binary code comfortably. Substantial portions of the ISA are ignored completely for the sake of simplicity. The notes use the assembly notation used by the GNU tools, including the assembler `as` (used by the compiler `gcc`) and the debugger `gdb`. Other tools may define other notations, but such things are merely cosmetic so long as you pay attention to what you are using at the time.

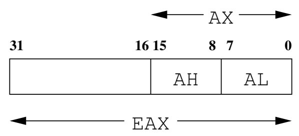
The Basics: Registers, Data Types, and Memory

You may have heard or seen the term "Reduced Instruction Set Computing," or RISC, and its counterpart, "Complex Instruction Set Computing," or CISC. While these terms were never entirely clear and have been further muddied by years of marketing, the x86 ISA is certainly vastly more complex than that of MIPS. On the other hand, much of the complexity has to do with backwards compatibility, which is mostly irrelevant to someone writing code today. Furthermore, we need use only a limited subset of the ISA in this class.

Modern flavors of x86—also called IA32, or Intel Architecture 32—have eight 32-bit integer registers. The registers are not entirely general-purpose, meaning that some instructions limit your choice of register operands to fewer than eight. A couple of other special-purpose 32-bit registers are also available—namely the instruction pointer (program counter) and the flags (condition codes), and we shall ignore the floating-point and multimedia registers. Unlike most RISC machines, the registers have names stemming from their historical special purposes, as described below.

`%eax`	accumulator (for adding, multiplying, *etc.*)
`%ebx`	base (address of array in memory)
`%ecx`	count (of loop iterations)
`%edx`	data (*e.g.*, second operand for binary operations)
`%esi`	source index (for string copy or array access)
`%edi`	destination index (for string copy or array access)
`%ebp`	base pointer (base of current stack frame)
`%esp`	stack pointer (top of stack)
`%eip`	instruction pointer (program counter)
`%eflags`	flags (condition codes and other things)

		8–bit	
32–bit	16–bit	high	low
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	DI		
EDI	DI		
EBP	SP		
ESP	BP		



The character "%" is used to denote a register in assembly code and is not considered a part of the register name itself; note also that register names are not case sensitive. The letter "E" in each name indicates that the "extended" version of the register is desired (extended from 16 bits). Registers can also be used to store 16- and 8-bit values, which is useful when writing smaller values to memory or I/O ports. As shown to the right above, the low 16 bits of a register are accessed by dropping the "E" from the register name, *e.g.*, `%si`. Finally, the two 8-bit halves of the low 16 bits of the first four registers can be used as 8-bit registers by replacing "X" with "H" (high) or "L" (low).

The x86 ISA supports both 2's complement and unsigned integers in widths of 32, 16, and 8 bits, single and double-precision IEEE floating-point, 80-bit Intel floating-point, ASCII strings, and binary-coded decimal (BCD). Most instructions are independent of data type, but some require that you select the proper instruction for the data types of the operands. Try multiplying 32-bit representations of -1 and 1 to produce a 64-bit result, for example.

Use of memory is more flexible in x86 than in MIPS: in addition to load and store operations, many x86 operations accept memory locations as operands. For example, a single instruction serves to read the value in a memory location, add a constant, and store the sum back to the memory location. With x86, memory is 8-bit (byte) addressable and uses 32-bit addresses, although few machines today fully populate this 4 GB address space.

One aspect of x86's treatment of memory may confuse you: *it is little endian*. Little endian means that if you store a 32-bit register into memory and then look at the four bytes of memory one by one, you will find the little end of the 32 bits first, followed by the next eight bits, then the next, and finally the high eight bits of the stored value. Thus 0x12345678 becomes 0x78, 0x56, 0x34, 0x12 in consecutive memory locations. Obviously, values read from memory also use this mapping, so that reading the bytes back in as a 32-bit value produces 0x12345678 again.

The x86 Instruction Set Architecture

You will find that there are many similarities between MIPS and x86. There are all of the same basic constructs: operate instructions, data movement instructions, and control flow instructions.

Operate instructions: Arithmetic operators are ADD, SUB, NEG (negate), INC (increment), and DEC (decrement); logical operators are AND, OR, XOR, and NOT; shift operators are SHL (left), SAR (arithmetic right), and SHR (logical right); finally, one can rotate bits, *i.e.*, shift with wraparound, to the left (ROL) or to the right (ROR). The first key difference is that x86 instructions typically specify one register as both the destination and one of the sources. Thus, one can execute

```
addl %eax,%ebx   # EBX ← EBX + EAX
```

but cannot use a single ADD instruction to put the sum of EAX and EBX into ECX. The part to the right in the example above is an x86 assembly language comment showing you the interpretation of the instruction in RTL, or register transfer language, with which you should already be familiar. Also, as you may have noticed from the example, the instruction name is extended with a label for the type of data, an "L" in the case above to indicate long, or 32-bit, operands. The other possibilities are "W" for 16-bit (word) and "B" for 8-bit (byte) operands. These markers are not required unless the operand types are ambiguous, but always using them can help to bring bugs to your attention.

Another difference that you might have noticed between MIPS assembly code and x86 assembly code—as defined by GNU's `as` in the latter case—is that the x86 destination register appears as the last operand rather than the first. Such orderings can be assembler-specific, but keep this ordering in mind when writing x86 assembly in this class.

Operate instructions in MIPS also allow the use of immediate values, but are usually restricted to values that fit in 16 bits. The x86 ISA uses variable-length instructions, so immediate values of up to 32-bits are usually allowed, and values that fit into fewer bits are encoded as shorter instructions. Immediate values are preceded with a dollar sign in the assembler, thus

```
addl $20,%esp   # ESP ← ESP + 20
```

adds 20 to the current value of ESP. Numbers starting with digits 1 through 9 are treated as decimal values; numbers starting with the prefix "0x" are treated as hexadecimal values; and numbers starting with the digit 0 (but no "x") are treated as octal values.

One aspect of x86's operand flexibility may end up causing problems for you. In particular, an operand may be either a memory reference or an immediate value. For example, removing the dollar sign from the stack addition above produces:

```
addl 20,%esp   # ESP ← ESP + M[20]
```

in which the contents of memory location 20 are added to ESP rather than the value 20. *Be careful to include a "$" when you want a number interpreted as an immediate value!* Note that labels are also affected in the same way with these instructions. A label without a preceding dollar sign generates a memory reference to the memory location marked by the label. A label preceded by a dollar sign results in an immediate operand with the value of the label (typically a 32-bit value).

One last comment about operate instructions. The x86 does support a fairly general addressing mode that can be used in combination with the LEA instruction (load effective address) to support more general addition operations. In particular, the format is

```
displacement(SR1,SR2,scale)
```

which multiplies `SR2` by `scale`, then adds both `SR1` and `displacement`. For example, one can use a single LEA instruction to put the sum of EAX and EBX into ECX as follows:

```
leal (%eax,%ebx),%ecx   # ECX ← EAX + EBX
```

In this case, both `displacement` and `scale` have been left off, in which case they default to zero and one, respectively. The original purpose and limitations of this addressing mode are discussed in the next section.

Finally, the list below shows a few common operations for which one used arithmetic instructions in MIPS.

```
movl  $10,%esi    # ESI ← 10
movl  %eax,%ecx   # ECX ← EAX
xorl  %edx,%edx   # EDX ← 0
```

Data movement instructions: MIPS is a load-store architecture, meaning that the only instructions that access memory are those that move data to and from registers. MIPS only provides one addressing mode for loads and stores: base_register+offset, generally written as `offset(base)`.

The x86 ISA supports both more opcodes and more addressing modes than the MIPS. Loads and stores in x86 are unified into the MOV instruction. However, as many x86 operations can use memory operands directly, not all data movement requires the use of MOV.

Most x86 addressing modes can be viewed as specific cases of the general mode described earlier (`displacement(SR1,SR2,scale)`). The purpose of this addressing mode was originally to support array accesses generated by high-level programs. For example, to access the N^{th} element of an array of 32-bit integers, one could put a pointer to the base of the array into EBX and the index N into ESI, then execute

```
movw (%ebx,%esi,4),%eax   # EAX ← M[EBX + ESI * 4]
```

If the array started at the 28th byte of a structure, and EBX instead held a pointer to the structure, one could still use this form by adding a displacement:

```
movw 28(%ebx,%esi,4),%eax   # EAX ← M[EBX + ESI * 4 + 28]
```

The `scale` can take the values one, two, four, and eight, and defaults to one. Examples of how one can use this mode and its simpler forms include the following:

```
movb  (%ebp),%al                # AL ← M[EBP]
movb  -4(%esp),%al              # AL ← M[ESP - 4]
movb  (%ebx,%edx),%al           # AL ← M[EBX + EDX]
movb  13(%ecx,%ebp),%al         # AL ← M[ECX + EBP + 13]
movb  (,%ecx,4),%al             # AL ← M[ECX * 4]
movb  -6(,%edx,2),%al           # AL ← M[EDX * 2 - 6]
movb  (%esi,%eax,2),%al         # AL ← M[ESI + EAX * 2]
movb  24(%eax,%esi,8),%al       # AL ← M[EAX + ESI * 8 + 24]
```

x86 also includes a direct addressing mode, in which the address to be used is specified as an immediate value in the instruction. The examples below show a number of ways in which data can be moved to and from specific address (often represented by labels in assembly code).

```
movb  100,%al                   # AL ← M[100]
movb  label,%al                 # AL ← M[label]
movb  label+10,%al              # AL ← M[label+10]
movb  10(label),%al             # NOT LEGAL!

movb  label(%eax),%al           # AL ← M[EAX + label]
movb  13+8*8-35+label(%edx),%al # AL ← M[EDX + label + 42]

movw  $label,%eax               # EAX ← label
movw  $label+10,%eax            # EAX ← label+10
movw  $label(%eax),%eax         # NOT LEGAL!
```

The middle two examples above return to the more general addressing mode and demonstrate some of the GNU assembler's capabilities in terms of evaluating expressions. Replacing MOVB with LEAL (and AL with EAX) in any of the first two groups of examples results in EAX being filled with the address used for the load. Putting an immediate marker ("$") in front of the label (as in the last group) has the same effect for some forms, but is not always legal.

Condition codes: For our purposes, the x86 has five relevant condition codes. The sign flag (SF) records whether the last result represented a negative 2's complement integer (had its most significant bit set). The zero flag (ZF) records whether the last result was exactly zero. The carry flag (CF) records whether the last result generated a carry or required a borrow, but is also used to hold bits shifted out with shifts, to record whether the high word of a multiplication is non-zero or not, and other such things (*i.e.*, there are far too many instruction-specific effects to list here). The overflow flag (OF) records whether the last operation overflowed when interpreted as a 2's complement operation, and serves additional purposes in the same fashion as CF. Finally, the parity flag (PF) records whether the last result had an even number of 1's or not; it is set for even parity, and clear for odd parity.

One aspect of the x86 ISA's use of flags is important to keep in mind when writing code: *not all result-producing instructions affect the flags, and not all flags are affected by instructions that affect some flags*. That said, most instructions mentioned so far affect all flags, The exceptions include MOV, LEA, and NOT, which affect no flags;

ROL and ROR, which affect only OF and CF; and INC and DEC, which affect all but CF.

In order to set the flags based on the result of a MOV or LEA (or any other instruction that doesn't affect the flags), use either a CMP (compare) or TEST instruction to set the flags first. The CMP instruction performs a subtraction, subtracting its first argument from its second, and sets the flags based on the result; nothing else is done with the result. The TEST instruction performs an AND operation between its two operations, sets the flags accordingly (OF and CF are cleared; SF, ZF, and PF are set according to the result), and discards the result of the AND.

Conditional branches: Eight basic branch conditions and their inverses are available with the x86 ISA, along with the unconditional branch JMP. These branches are described below, along with the conditions under which the branch is taken.

`jo`	overflow	OF is set	`jb`	below	CF is set
`jp`	parity	PF is set (even parity)	`jbe`	below or equal	CF or ZF is set
`js`	sign	SF is set (negative)	`jl`	less	SF $\neq$ OF
`je`	equal	ZF is set	`jle`	less or equal	(SF $\neq$ OF) or ZF is set

The sense of each branch other than JMP can be inverted by inserting an "N" after the initial "J," *e.g.*, JNB jumps if the carry flag is clear. Furthermore, many of the branches have several equivalent names. For example, JZ (jump if zero) can be written in place of JE (jump if equal). Unsigned comparisons should use the "above" and "below" branches, while signed comparisons should use the "less" or "greater" branches, as shown in the table below. The preferred forms are those that the debugger uses when disassembling code.

	`jnz`	`jnae`	`jna`	`jz`	`jnb`	`jnbe`	unsigned comparisons
preferred form	`jne`	`jb`	`jbe`	`je`	`jae`	`ja`	
	$\neq$	$<$	$\leq$	$=$	$\geq$	$>$	
preferred form	`jne`	`jl`	`jle`	`je`	`jge`	`jg`	signed comparisons
	`jnz`	`jnge`	`jng`	`jz`	`jnl`	`jnle`	

The table should be used as follows. After a comparison such as

```
cmp %ebx,%esi   # set flags based on (ESI - EBX)
```

choose the operator to place between ESI and EBX, based on the data type. For example, if ESI and EBX hold unsigned values, and the branch should be taken if ESI $\leq$ EBX, use either JBE or JNA. If ESI and EBX hold signed values, and the branch should be taken if ESI $>$ EBX, use either JG or JNLE. For branches other than JE/JNE based on instructions other than CMP, you should check the stated branch conditions rather than trying to use the table.

Other control instructions: Subroutine control is straightforward. The CALL instruction in x86 plays the role of the MIPS subroutine call instructions, `jal` and `jalr`. The single operand of a CALL can be either direct or indirect; indirect operands are preceded by an asterisk:

```
call  printf            # (push EIP), EIP ← printf
call  *%eax             # (push EIP), EIP ← EAX
call  *(%eax)           # (push EIP), EIP ← M[EAX]
call  *fptr             # (push EIP), EIP ← M[fptr]
call  *10(%eax,%edx,2)  # (push EIP), EIP ← M[EAX + EDX*2 + 10]
```

The CALL instruction pushes the return address onto the stack before changing the instruction pointer. Its counterpart, the RET (return) instruction, then pops the return address off the stack and into EIP in order to return from the called routine. The calling convention is described in greater detail later in these notes.

The unconditional branch instruction JMP mentioned earlier also takes the role of the indirect jump instruction in x86, using the same syntax as shown above for the CALL instruction.

Two types of instructions are not covered by these notes. The x86 INT instruction provides support for system calls through a mechanism similar to the MIPS `syscall` instruction, except the x86 version passes a value. However, understanding its use in a more modern operating system requires some discussion. Similarly, x86's return from interrupt (IRET) instruction is similar to MIPS's `rfe`.

Labels, comments, directives, and pseudo-ops: Labels can begin with any letter, a period, or an underscore. Characters after the first can also include numbers. Later characters can also include dollar signs, but introducing the context-specific meaning can be confusing. Is it part of a label, or an immediate value marker? *Each label definition must be followed by a colon*; uses of a label do not include this colon, and it is not considered to be part of the label. *Labels are case sensitive.* Labels are the only case sensitive aspect of the `as` assembler mentioned in these notes. Finally, if you look at assembly code generated by the `gcc` compiler, you will notice that it starts its label names with a period; if you want to mix your code with code generated from C, you may want to avoid starting your labels with periods.

Comments can take two forms. The examples in these notes so far have used a form similar to that found in the MIPS assembler. In particular, the assembler ignores everything on a given line after the first pound sign ('#"). Multi-line comments are also allowed with `as`, using C-style demarcation:

```
/* A comment of this form
   can span multiple lines.  */
```

Note that semicolons can be used to separate x86 instructions grouped onto a single line, as shown here:

```
movw my_data_ptr,%eax ; movw (%eax),%eax   # EAX ← M[M[my_data_ptr]]
```

The following assembler directives are all fairly useful. The x86 assembler supports both .GLOBAL and .EXTERN to declare symbols to be visible externally and to be defined externally, respectively. The .EXTERN directive is technically unnecessary: the assembler assumes that any undefined symbol is defined elsewhere. *This assumption implies that the assembler cannot identify undefined symbols until link time!* Take the time to figure out what a missing symbol looks like as a linker error in advance so that you don't have to guess when you see the error later.

The MIPS `.asciiz` directive becomes .STRING. Empty space can be created using the .SPACE directive, which requires a first argument specifying the number of bytes and accepts an optional second argument specifying the byte value to use as filler. Examples of these new directives are shown below:

```
.byte    12,-15      # 8-bit values
.word    200,4000    # 16-bit values
.long    -987654321  # 32-bit values
.quad    9999999999  # 64-bit values
.single  1.0,2.0     # single-precision IEEE floating-point
.double  2.0,3.1415  # double-precision IEEE floating-point

# alternate forms
.hword   22000,-17   # 16-bit values
.int     1,4,9,0x16  # 32-bit values
.float   2.7         # single-precision IEEE floating-point
```

Another useful directive is worth mentioning here: .INCLUDE. This directive tells the assembler to read in the contents of another file and to insert it in place of the directive. It is equivalent to the C preprocessor's `#include` directive.

Input and output: MIPS uses memory-mapped I/O for all devices, as do most modern architectures. In contrast, older architectures like the x86 were originally designed with separate I/O name spaces and special instructions for accessing them. Originally, the 8086 communicated only through devices such as the serial port, to which one could attach a terminal for displaying the output and a keyboard for driving the input to the processor. Someone soon realized that one could drop a display card with memory into the system bus and take over part of the memory's address space without changing the processor, however, and ever since, x86-based desktop computers have used both memory-mapped and instruction-based I/O.

The I/O instructions have not changed substantially since the original 8086 ISA, and require the use of specific registers. In particular, while one can now write a 32-bit word to a sequence of I/O ports, the data must still be in the EAX register (AX for 16-bit, or AL for 8-bit). Data from ports are also read into EAX. Similarly, the port number to be used can be specified as either an 8-bit immediate or loaded into DX (the port space is 16-bit). The examples below use the notation P[x] to denote port x.

```
inb   $0x40,%al   # AL ← P[0x40]
inw   (%dx),%ax   # AL ← P[DX], AH ← P[DX + 1]
outb  %al,(%dx)   # P[DX] ← AL
outw  %ax,68      # P[68] ← AL, P[69] ← AH
```

As illustrated explicitly in the examples above, the port address space is treated much like memory with x86. In particular, it is both byte-addressable and little endian. Thus writing 16 bits to a port writes the low 8 bits to the named port, and the high 8 bits to the next port. Finally, like MOV, neither IN nor OUT affects the flags.

Other Useful Instructions

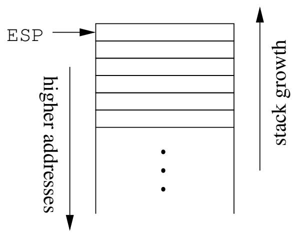
The last section focused primarily on x86 instructions similar to those available in MIPS. This section introduces a few types of instructions that require more substantial software support with MIPS.

Stack operations: The x86 ISA supports a stack abstraction directly rather than as a software convention. The PUSH and POP instructions provide the necessary functionality. As shown to the right, the stack convention used is that ESP contains the address of the element on top of the stack. The stack grows downward in addresses, like that of the MIPS, so

```
     pushl %eax   # M[ESP - 4] ← EAX, ESP ← ESP - 4
```

is equivalent to

```
         movl  %eax,-4(%esp)  # M[ESP - 4] ← EAX
         subl  $4,%esp        # ESP ← ESP - 4
```



Other than a POP into the EFLAGS register, PUSH and POP do not affect the flags.

Multiplication and division: Signed and unsigned forms of integer multiplication and division are available in the x86 ISA. The unsigned multiply (MUL) requires that EAX be one of the operands (or AX, or AL), and places the high bits of the result in EDX, and the low bits in EAX (or DX:AX, or AX). Both signed (IDIV) and unsigned (DIV) division have similar restrictions. The dividend must be placed in EDX:EAX (or DX:AX, or AX). After an IDIV instruction, EAX (or AX, or AL) holds the quotient, and EDX (or DX, or AH) holds the remainder. If the quotient overflows the destination register, an exception is generated.

Signed multiplication is more flexible. Although the instruction formats supported with unsigned multiplication are also available, signed multiplication also allows two- and three-operand forms, as shown below. In these forms, the high bits of the product are discarded.

```
imull  %ebx,%eax         # EAX ← EAX * EBX
imull  $1000,%ebx,%eax   # EAX ← 1000 * EBX
```

The flags are undefined after division, and only the CF and OF flags have a meaning with multiplication (note that the other flags are undefined, not unaffected; do not expect them to retain their previous values). With multiplication, both CF and OF are set whenever the high bits of the result are non-zero.

Data type conversions: The MOV instruction can also be used to convert small integers into larger ones through sign or zero extension. Converting large integers into smaller ones is usually done by simply using other register names (*e.g.*, AX or AL for a value in EAX). After MOV, add either "S" for sign extension or "Z" for zero extension, then a letter for the original size, and a letter for the final size. For example, MOVZBL zero extends a byte from memory or another register into a long (32 bits). Special forms are available for EAX: CBTW converts signed byte AL to word AX, CWTL converts signed word AX to long word EAX, and CLTD converts signed long word EAX to double word EDX:EAX. The last is useful in preparing for IDIV. Be careful about CWTD: it exists, but changes AX into DX:AX.

The Calling Convention

Writing x86 assembly that interfaces with high-level languages such as C requires that one understand the calling convention for the ISA. This section begins with a description of the rules for passing and returning values and register ownership in the x86 calling convention, then provides an example in which a C function and a use of that function are translated into x86 assembly code.

Parameters, return values, and registers: Parameters passed to a function are pushed onto the stack with x86. In most high-level languages, parameters are pushed from right to left to allow for a variable number of parameters without requiring additional space for parameter counts or sentinels.

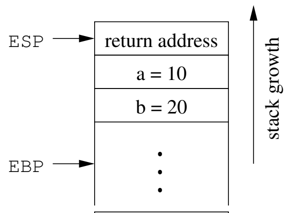
The location of the value returned from a function depends on the type of the value being returned. For pointers and integers requiring no more than 32 bits, the return value is placed in EAX. Integers and other non-floating-point types of more than 32 but no more than 64 bits are split between EDX (high bits) and EAX (low bits). Floating-point values are returned on the top of the floating-point stack (not discussed in these notes).

Most of the registers are considered to be owned by the caller. Both the stack and the frame pointer, ESP and EBP, must be returned unchanged. Similarly, EBX, ESI, and EDI are callee-saved. The return values EAX and EDX must obviously be saved by the caller, if they are to be preserved. The ECX register is also caller-saved (as is EFLAGS).

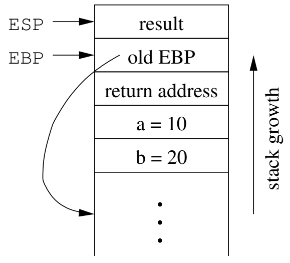
```
/* call the function */    # the call site...
value = a_func (10, 20);      pushl  $20             # push second argument
                              pushl  $10             # push first argument
                              call   a_func          # call the function
                              addl   $8,%esp         # pop the arguments
                              movl   %eax,-4(%ebp)   # store result in 'value'

int                        a_func:
a_func (int a, int b)         pushl  %ebp            # save old frame pointer
{                             movl   %esp,%ebp       # point to new frame
   int result;                subl   $4,%esp         # make room for 'result'
                              movl   8(%ebp),%eax    # put 'a' into EAX
   result = a * b + 1;        imull  12(%ebp),%eax   # multiply EAX by 'b'
                              incl   %eax            # add one
   return result;             movl   %eax,-4(%ebp)   # store into 'result'
}                             movl   -4(%ebp),%eax   # return 'result' in EAX
                              leave                  # restore frame pointer
                              ret                    # return
```

Caller side: Consider the code shown above in which the C function `a_func` is called by another piece of code. The C versions of the call site and the function appear on the left, with their translations to the right. For clarity, the assembly code is not optimized.

To prepare for a call, the caller first pushes the function parameters onto the stack. In this case, the immediate values 20 and 10 are pushed. The CALL instruction is then executed, which pushes the next EIP (pointing to the ADD after the CALL) onto the stack and changes the EIP to the start of the function. The diagram to the right illustrates the stack pointer values at the start of the function. The top of the stack (ESP) holds the return address, under which are the function parameters (often called **formals** within the function). The EBP register still points to the stack frame of the caller. The function `a_func` then executes as described in the next section, returning the two stack registers to the values shown at the right before executing the RET instruction. The RET instruction pops the return address from the stack. The caller then removes the parameters from the stack using an ADD and stores the return value from EAX into the appropriate local variable (relative to its own frame pointer).



Callee side: The instructions in the function can be broken into three groups: one to set up the function's stack frame, a second to implement the function, and a third to tear down the stack frame. The state of the stack during function execution is shown in the diagram to the right, so the stack frame set up code shifts from the diagram above to the one to the right, and the tear down code returns the stack state to that shown above.



Set up begins by pushing the old frame pointer (EBP) onto the stack, then copying ESP into EBP. A copy of the old frame pointer thus sits at the base of the new stack frame, and the new frame pointer points to it, effectively forming a linked list of stack frames. If any callee-saved registers (EBX, ESI, and EDI) are used by the function implementation, they are then pushed onto the stack (none are used by `a_func`). Finally, the stack pointer is updated to make room for local variables.

Tearing down the stack frame requires only a single instruction when no callee-saved registers have been preserved. The LEAVE instruction restores the old values of EBP and ESP (in RTL, `ESP` ← `EBP + 4`, `EBP` ← `M[EBP]`). When callee-saved registers must be restored, an LEA instruction points the ESP to the uppermost saved register, then a sequence of POPs (the last into EBP) restores the original stack state.

Miscellany

Data type alignment: Most architectures support byte-addressable memory. However, when moving data to and from memory, they often restrict the address used to even multiples of the data type. For example, 16-bit values can only be written or read from even addresses, and 32-bit values require addresses that are multiples of four (32 bits is four bytes). The x86 ISA technically does not require such alignment, but only because the 8086 did not impose any restrictions. From a performance standpoint, however, unaligned loads and stores are extremely slow. The rationale is that anyone running software old enough to contain unaligned accesses could not possibly care about the performance of that software, given that the clock speed of the processor is already three orders of magnitude faster.

The implication for your programs is that arrays of data (as well as code, but for slightly different reasons) should be properly aligned in memory. To ensure proper alignment, use the .ALIGN directive, which takes a single argument and inserts enough blank space to reach the next address that is a multiple of the argument. For example, if you want to declare an array of 32-bit numbers, you should write something similar to the following:

```
            .align  4  # the label my_array (an address) is a multiple of 4
my_array:   .long   100000,4000000000,24,0
```

Support for larger data types: Larger integer data types can be constructed from the existing types by operating on the larger values word by word and using the carry flag to simulate larger adders, multipliers, *etc.*. For example, to add two 64-bit numbers, first add the low 32 bits with ADDL, then use ADCL (add with carry, long) to add the high 32 bits along with the carry flag. If you are interested in such things, take a look at add with carry (ADC), subtract with borrow (SBB), and rotate with carry (RCL/RCR).

Getting More Information

Many sources of information are available if you want to learn more about the x86 ISA or implementations thereof. You can find ISA manuals (from Intel, or AMD for AMD-64) online; official manuals for the x86 family are available at `http://www.x86.org/intel.doc/` and many informal guides are available elsewhere. The mnemonics and syntax may differ, but the instruction sets are the same, so you should be able to figure out how to get an instruction to work once you know that it exists.

For information about the GNU assembler, the best source is the info page: type “info as” at the prompt in Linux; the info interface is much like that of `emacs`, so hopefully you’re comfortable with that interface. For many instructions, calling conventions, function prototypes, *etc.*, you can simply write the code you want in C and have gcc compile it to assembly for you with the -S flag. Don’t use -o with -S; by default, gcc will create a new file ending in .s, but you can (and should not) override it with -o. Also beware of clobbering a modified .s file! Much of the information in this set of notes was gathered with -S.

Finally, you may find Randy Hyde’s e-books on programming in x86 assembly useful. They are freely available from his web site: `http://webster.cs.ucr.edu`.