**8086 Assembly Language**

**Six Things You Should Know About the 8086**

**1) The 8086 is a 16-bit processor.**

Because the word size is limited to 16-bits, many data types in C have different sizes then they do on the spice machines or modern PCs. Below is a list of C data types and their sizes.

Type | Length | Range

---------------|---------|-----------------------------------------

unsigned char | 8 bits | 0 to 255

char | 8 bits | -128 to 127

enum | 16 bits | -32,768 to 32,767

unsigned int | 16 bits | 0 to 65,535

short int | 16 bits | -32,768 to 32,767

int | 16 bits | -32,768 to 32,767

unsigned long | 32 bits | 0 to 4,294,967,295

long | 32 bits | -2,147,483,648 to 2,147,483,647

Most importantly, note the range limitation on the int data type. It is also important to realize that any operations on 32-bit data types (long and unsigned long) require from several to very many instructions to perform operations. 32-bit operations in C should be avoided unless absolutely necessary.

The following names are used to refer to data sizes on the 8086:

Length | Size Name

--------|-----------------------

4-bit | nibble

8-bit | byte

16-bit | word

32-bit | dword (or doubleword)

**2) The 8086 uses little endian format.**

This means that the least significant byte of a value is stored first (i.e., at the low address) in memory. This gives the appearance of numbers being stored in memory backwards. For example, the 32-bit value 0x11223344 would be stored as bytes in the following order:

(low addr) (high addr)

0x44 0x33 0x22 0x11

This must be kept in mind when accessing different parts of data.

**3) The 8086 uses segmented memory.**

A memory address on the 8086 consists of two numbers, usually written in hexadecimal and separated by a colon, representing the *segment* and the *offset*. This combination of segment and offset is referred to as a *logical address*. The segment number refers to a 64 KB block of memory and the offset is an index into the memory segment. For example the address AB10:1024 corresponds to the byte stored in segment 0xAB10 at offset 0x1024.

Both the segment and offset are represented by a 16-bit number, allowing each segment to be 2^16 bytes in size (i.e., 65536 bytes, or 64 KB). This would seem to suggest that the 8086 can address up to 2^32 bytes, or 4 GB, since 32 bits are used for each address. This is NOT the case.

When the processor obtains a logical address (segment and offset), it performs a simple calculation to determine the 20-bit *physical address* in memory to which the logical address refers:

physical address = (segment << 4) + offset

This is equivalent to multiplying the segment by 16 and adding the offset (i.e., physical address = segment \* 16 + offset). This means that the 64 KB segments overlap, with a new segment starting every 16 bytes. This also means that there can be more than one address for the same memory location. For example, 0000:0100, 0001:00F0, and 0010:0000 all refer to physical address 0x100. There are even more examples we could give for that same memory location.

In this class, you will not usually need to worry about segments because your programs will only deal with the first segment of memory. In this case, you can think of memory as a single continuous piece of memory that is 64 KB in size.

**4) Registers in the 8086 have intended uses.**

The 8086 has four 16-bit general purpose registers, five 16-bit offset registers for accessing memory, four 16-bit segment registers also for memory access, and a 16-bit flags register. Nine bits of the flags register are accessible to the programmer and each of these bits is referred to as a flag. Each flag either indicates a condition or controls the behavior of certain instructions. For example, the cmp instruction compares two numbers and sets flags based on the relationship between these numbers. Other instructions, such as je (for jump if equal), can then be used, behaving differently depending on the state of the flags previously set by the cmp instruction.

Most instructions only allow certain registers to be used as operands and some instructions require specific registers to be used. Therefore, it is important to be familiar with the different registers and their intended purposes. However, there is still a lot of freedom in what registers can be used. Below is a list of the 8086 registers. This listing can also be obtained in Emu86 by entering "regs". For each register the assembly symbol, name, and intended use are given.

General Purpose Registers (a.k.a. scratch registers)

AX (AH,AL) Accumulator : Main arithmetic register

BX (BH,BL) Base : Generally used as a memory base or offset

CX (CH,CL) Counter : Generally used as a counter for loops

DX (DH,DL) Data : General 16-bit storage, division remainder

Offset Registers

IP Instruction pointer : Current instruction offset

SP Stack pointer : Current stack offset

BP Base pointer : Base for referencing values stored on stack

SI Source index : General addressing, source offset in string ops

DI Destination index : General addressing, destination in string ops

Segment Registers

CS Code segment : Segment to which IP refers

SS Stack segment : Segment to which SP refers

DS Data segment : General addressing, usually for program's data area

ES Extra segment : General addressing, destination segment in string ops

Flags Register (Respectively bits 11,10,9,8,7,6,4,2,0)

OF Overflow flag : Indicates a signed arithmetic overflow occurred

DF Direction flag : Controls incr. direction in string ops (0=inc, 1=dec)

IF Interrupt flag : Controls whether interrupts are enabled

TF Trap flag : Controls debug interrupt generation after instructions

SF Sign flag : Indicates a negative result or comparison

ZF Zero flag : Indicates a zero result or an equal comparison

AF Auxiliary flag : Indicates adjustment is needed after BCD arithmetic

PF Parity flag : Indicates an even number of 1 bits

CF Carry flag : Indicates an arithmetic carry occurred

The general purpose registers AX, BX, CX, and DX are 16-bit registers but each can also be used as two separate 8-bit registers. For example, the high (or upper) byte of AX is called AH and the low byte is called AL. The same H and L notation applies to the BX, CX, and DX. Most instructions allow these 8-bit registers as operands.

Registers AX, BX, CX, DX, SI, DI, BP, and SP can be used as operands for most instructions. However, only AX, BX, CX, and DX should be used for general purposes since SI, DI, BP, and SP are usually used for addressing.

**5) The 8086 instructions can use register, immediate, and memory operands.**

The 8086 is not limited to immediate or register operands. Most instructions also allow memory operands to be used. For example, if a word sized variable were pointed to by the value stored in register BX, the number 3 could be added to it using the following instruction:

add word [bx], 3

The brackets indicate that BX is to be used as a pointer to a memory location. The only limitation is that there can be only one memory reference per instruction. For example, the following addition instruction is invalid:

add word [bx], word [si] ; Bad instruction!

Instead you would use two instructions:

mov ax, [si] ; Load [si] into ax

add [bx], ax ; Add to [bx]

**6) The 8086 is the ancestor of modern Intel processors.**

8086 code runs fine on modern x86 processors, such as the Pentium processors. However, modern x86 code rarely runs on an 8086. When experimenting with 8086 assembly language code, be careful to check the processor on which instructions work. Many instructions have been added since the 8086 was first produced so instructions for newer processors must be avoided. The documentation for this class only covers 8086 instructions.

**Referencing Memory**

**Segment and Offset**

Recall that the 8086 uses logical addresses composed of a *segment* and an *offset* to reference memory. Every memory reference on the 8086 will use one of the segment registers (i.e., DS, ES, SS, CS, or SS) as the segment combined with an offset (usually given in the instruction) to determine the physical address being referenced. The physical address referenced is always

physical address = (segment << 4) + offset.

**The Effective Address**

There are several ways to reference memory locations and specific registers that must be used. A memory reference is placed in brackets to distinguish it from a register or immediate value. In general, memory accesses take the form of the following example:

mov ax, [*baseReg* + *indexReg* + *constant*]

This example copies a word sized value into the register AX. Combined, the three parameters in brackets determine what is called the *effective address*, which is simply the offset referenced by the instruction. The following rules apply:

*baseReg* can be: bp or bx

*indexReg* can be: si or di

*constant* can be: 16-bit signed number if combined with registers, as in "mov ax,[bp+2]"

16-bit unsigned number if by itself, as in "mov ax,[2]"

Any one or two of the memory access parameters (i.e., constant, baseReg, or indexReg) can be omitted, allowing for several memory access modes.

It is important to realize that the effective address, or offset, does NOT give the complete address for the memory reference. A segment register is either implied or given in the instruction. This topic is discussed in the section *Segment Registers* below.

**Segment Registers**

One of the segment registers is always used as the segment when evaluating an address. The available segment registers are the *Data Segment* (DS), *Extra Segment* (ES), *Stack Segment* (SS), or *Code Segment* (CS). Therefore, you must be aware of which segment register is used when an address is evaluated as part of an instruction. When a memory reference is given in an instruction, the processor sums any baseReg, indexReg, and constant that are given and uses this sum as the offset into the segment.

Which segment register that is used in the address calculation depends on the register that is used for baseReg. The DS register is assumed for the segment unless baseReg is the register BP, in which case SS is assumed. However, any segment register can be explicitly specified using what is called a *segment override* prefix (discussed below). Also, some special instructions may assume other segment registers.

**Segment Overrides**

A segment override prefix allows any segment register (DS, ES, SS, or CS) to be used as the segment when evaluating addresses in an instruction. An override is made by adding the segment register plus a colon to the beginning of the memory reference of the instruction as in the following examples:

mov ax, [es:60126] ; Use es as the segment

mov ax, [cs:bx] ; Use cs as the segment

mov ax, [ss:bp+si+3] ; Use ss as the segment

**Operand Size**

A memory reference can be used as a source or destination operand for most 8086 instructions. Any time a memory reference is given as part of an instruction, the size of the memory operand is either implied or must be specified. For example consider the following instruction:

mov ax, [bx]

This instruction will move the word stored at DS:BX and put it into AX. The size of word is implied since the AX register is one word in size. In some cases the size of the operand must be given in order for the assembler to generate an instruction. For example, to increment a variable pointed to by BX, the assembler will not accept the following:

inc [bx] ; WRONG!!

This is because it does not know if [bx] addresses a byte or word sized value. So the size of [bx] must be specified, as in the following two examples:

inc word [bx] ; Increment word at [bx]

inc byte [bx] ; Increment byte at [bx]

It is not necessary to specify the size if one of the operands has a known size, such as a register operand, as in:

add al, [bx] ; Assembler knows al is a byte so "byte [bx]" is assumed

**Addressing Modes**

Here are some examples of the allowed addressing modes:

xor cx, [59507] ; Direct mode (XOR CX with word at **DS:E873**)

push word [bx] ; Register-indirect mode (Push word at **DS:BX** onto stack)

mov ax, [bp-4] ; Base mode (Move word at **SS:(BP-4)** into AX)

sub [si+2], bx ; Indexed mode (Subtract BX from word at **DS:(SI+2)**)

not byte [bp+di] ; Base-indexed mode (Invert bits of byte at **SS:(BP+DI)**)

add [bx+si+2], dx ; Base-indexed mode with dispacement (Add DX to word at **DS:(BX+SI+2)**)

The five addressing modes available are outlined more precisely for your reference below:

Direct Mode: [constant]

constant: 16-bit unsigned value

Register-Indirect Mode: [register]

register: bx, si, or di

Note: bp technically isn't allowed. If used, assembler will generate [bp+0] instead.

Base Mode: [constant + baseReg]

constant: 8-bit or 16-bit signed value

baseReg: bp or bx

Indexed Mode: [constant + indexReg]

constant: 8-bit or 16-bit signed value

indexReg: si or di

Base-Indexed Mode: [baseReg + indexReg]

baseReg: bp or bx

indexReg: si or di

Base-Indexed Mode with Displacement: [constant + baseReg + indexReg]

constant: 8-bit or 16-bit signed value

baseReg: bp or bx

indexReg: si or di

**Important Usage Notes:**

1. The first operand of an instruction is also the destination if there is a resulting value. Divide and multiply instructions are common exceptions to this rule.
2. There can be *at most* one memory operand per instruction.
3. There can be *at most* one immediate operand per instruction.
4. Operands generally must be of the same size (i.e., byte or word).
5. Using a label is the same as using an immediate or constant value.
6. When BP is used in a memory reference, SS is assumed as the segment. Otherwise DS is assumed.
7. While an instruction is executing, IP refers to the next instruction.
8. Many instructions are smaller if you use the appropriate registers (usually AX or AL).
9. In NASM, all labels are case sensitive but instruction and register names are not.

**Terminology Used:**

* **memory** - Refers to an 8 or 16-bit memory location determined by an effective address.
* **register** - AX, BX, CX, DX, SI, DI, BP, or SP as well as the 8-bit derivatives of AX, BX, CX, and DX (other registers or flags are not allowed).
* **immediate** - A numeric constant or label.
* **REG1::REG2** - The concatenation of two registers (e.g., the 32-bit value DX::AX) A single colon is used for memory addresses.
* **XF** or **XF=b** - A flag's value after an instruction can be 0 or 1 and usually depends on the result of the instruction. A flag being set to '?' by an instruction indicates that the flag is undefined after the operation.

**Instructions:**

**adc Add with carry flag**

Syntax: adc dest, src

dest: memory or register

src: memory, register, or immediate

Action: dest = dest + src + CF

Flags Affected: OF, SF, ZF, AF, PF, CF

Notes: This instruction is used to perform 32-bit addition.

**add Add two numbers**

Syntax: add dest, src

dest: register or memory

src: register, memory, or immediate

Action: dest = dest + src

Flags Affected: OF, SF, ZF, AF, PF, CF

Notes: Works for both signed and unsigned numbers.

**and Bitwise logical AND**

Syntax: and dest, src

dest: register or memory

src: register, memory, or immediate

Action: dest = dest & src

Flags Affected: OF=0, SF, ZF, AF=?, PF, CF=0

**call Call procedure or function**

Syntax: call addr

addr: register, memory, or immediate

Action: Push IP onto stack, set IP to addr.

Flags Affected: None

**cbw Convert byte to word (signed)**

Syntax: cbw

Action: Sign extend AL to create a word in AX.

Flags Affected: None

Notes: For unsigned numbers use "mov ah, 0".

**cli Clear interrupt flag (disable interrupts)**

Syntax: cli

Action: Clear IF

Flags Affected: IF=0

**cmp Compare two operands**

Syntax: cmp op1, op2

op1: register or memory

op2: register, memory, or immediate

Action: Perform op1-op2, discarding the result but setting the flags.

Flags Affected: OF, SF, ZF, AF, PF, CF

Notes: Usually used before a conditional jump instruction.

**cwd Convert word to doubleword (signed)**

Syntax: cwd

Action: Sign extend AX to fill DX, creating a dword contained in DX::AX.

Flags Affected: None

Notes: For unsigned numbers use "xor dx, dx" to clear DX.

**dec Decrement by 1**

Syntax: dec op

op: register or memory

Action: op = op - 1

Flags Affected: OF, SF, ZF, AF, PF

**div Unsigned divide**

Syntax: div op8

div op16

op8: 8-bit register or memory

op16: 16-bit register or memory

Action: If operand is op8, unsigned AL = AX / op8 and AH = AX % op8

If operand is op16, unsigned AX = DX::AX / op16 and DX = DX::AX % op16

Flags Affected: OF=?, SF=?, ZF=?, AF=?, PF=?, CF=?

Notes: Performs both division and modulus operations in one instruction.

**idiv Signed divide**

Syntax: idiv op8

idiv op16

op8: 8-bit register or memory

op16: 16-bit register or memory

Action: If operand is op8, signed AL = AX / op8 and AH = AX % op8

If operand is op16, signed AX = DX::AX / op16 and DX = DX::AX % op16

Flags Affected: OF=?, SF=?, ZF=?, AF=?, PF=?, CF=?

Notes: Performs both division and modulus operations in one instruction.

**imul Signed multiply**

Syntax: imul op8

imul op16

op8: 8-bit register or memory

op16: 16-bit register or memory

Action: If operand is op8, signed AX = AL \* op8

If operand is op16, signed DX::AX = AX \* op16

Flags Affected: OF, SF=?, ZF=?, AF=?, PF=?, CF

**in Input (read) from port**

Syntax: in AL, op8

in AX, op8

op8: 8-bit immediate or DX

Action: If destination is AL, read byte from 8-bit port op8.

If destination is AX, read word from 16-bit port op8.

Flags Affected: None

**inc Increment by 1**

Syntax: inc op

op: register or memory

Action: op = op + 1

Flags Affected: OF, SF, ZF, AF, PF

**int Call to interrupt procedure**

Syntax: int imm8

imm8: 8-bit unsigned immediate

Action: Push flags, CS, and IP; clear IF and TF (disabling interrupts); load

word at address (imm8\*4) into IP and word at (imm8\*4 + 2) into CS.

Flags Affected: IF=0, TF=0

Notes: This instruction is usually used to call system routines.

**iret Interrupt return**

Syntax: iret

Action: Pop IP, CS, and flags (in that order).

Flags Affected: All

Notes: This instruction is used at the end of ISRs.

**j?? Jump if ?? condition met**

Syntax: j?? rel8

rel8: 8-bit signed immediate

Action: If condition ?? met, IP = IP + rel8 (sign extends rel8)

Flags Affected: None

Notes: Use the cmp instruction to compare two operands then j?? to jump

conditionally. The ?? of the instruction name represents the jump

condition, allowing for following instructions:

ja jump if above, unsigned >

jae jump if above or equal, unsigned >=

jb jump if below, unsigned <

jbe jump if below or equal, unsigned <=

je jump if equal, ==

jne jump if not equal, !=

jg jump if greater than, signed >

jge jump if greater than or equal, signed >=

jl jump if less than, signed <

jle jump if less than or equal, signed <=

All of the ?? suffixes can also be of the form n?? (e.g., jna for

jump if not above). See 8086 documentation for many more ?? conditions.

An assembler label should be used in place of the rel8 operand. The

assembler will then calculate the relative distance to jump.

Note also that rel8 operand greatly limits conditional jump distance

(-127 to +128 bytes from IP). Use the jmp instruction in combination

with j?? to overcome this barrier.

**jmp Unconditional jump**

Syntax: jump rel

jump op16

jump seg:off

rel: 8 or 16-bit signed immediate

op16: 16-bit register or memory

seg:off: Immediate 16-bit segment and 16-bit offset

Action: If operand is rel, IP = IP + rel

If operand is op16, IP = op16

If operand is seg:off, CS = seg, IP = off

Flags Affected: None

Notes: An assembler label should be used in place of the rel8 operand. The

assembler will then calculate the relative distance to jump.

**lea Load effective address offset**

Syntax: lea reg16, memref

reg16: 16-bit register

memref: An effective memory address (e.g., [bx+2])

Action: reg16 = address offset of memref

Flags Affected: None

Notes: This instruction is used to easily calculate the address of data in

memory. It does not actually access memory.

**mov Move data**

Syntax: mov dest, src

dest: register or memory

src: register, memory, or immediate

Action: dest = src

Flags Affected: None

**mul Unsigned multiply**

Syntax: mul op8

mul op16

op8: 8-bit register or memory

op16: 16-bit register or memory

Action: If operand is op8, unsigned AX = AL \* op8

If operand is op16, unsigned DX::AX = AX \* op16

Flags Affected: OF, SF=?, ZF=?, AF=?, PF=?, CF

**neg Two's complement negate**

Syntax: neg op

op: register or memory

Action: op = 0 - op

Flags Affected: OF, SF, ZF, AF, PF, CF

**nop No operation**

Syntax: nop

Action: None

Flags Affected: None

**not One's complement negate**

Syntax: not op

op: register or memory

Action: op = ~op

Flags Affected: None

**or Bitwise logical OR**

Syntax: or dest, src

dest: register or memory

src: register, memory, or immediate

Action: dest = dest | src

Flags Affected: OF=0, SF, ZF, AF=?, PF, CF=0

**out Output (write) to port**

Syntax: out op, AL

out op, AX

op: 8-bit immediate or DX

Action: If source is AL, write byte in AL to 8-bit port op.

If source is AX, write word in AX to 16-bit port op.

Flags Affected: None

**pop Pop word from stack**

Syntax: pop op16

reg16: 16-bit register or memory

Action: Pop word off the stack and place it in op16 (i.e., op16 = [SS:SP]

then SP = SP + 2).

Flags Affected: None

Notes: Pushing and popping of SS and SP are allowed but strongly discouraged.

**popf Pop flags from stack**

Syntax: popf

Action: Pop word from stack and place it in flags register.

Flags Affected: All

**push Push word onto stack**

Syntax: push op16

op16: 16-bit register or memory

Action: Push op16 onto the stack (i.e., SP = SP - 2 then [SS:SP] = op16).

Flags Affected: None

Notes: Pushing and popping of SS and SP are allowed but strongly discouraged.

**pushf Push flags onto stack**

Syntax: pushf

Action: Push flags onto stack as a word.

Flags Affected: None

**ret Return from procedure or function**

Syntax: ret

Action: Pop word from stack and place it in IP.

Flags Affected: None

**sal Bitwise arithmetic left shift (same as shl)**

Syntax: sal op, 1

sal op, CL

op: register or memory

Action: If operand is 1, op = op << 1

If operand is CL, op = op << CL

Flags Affected: OF, SF, ZF, AF=?, PF, CF

**sar Bitwise arithmetic right shift (signed)**

Syntax: sar op, 1

sar op, CL

op: register or memory

Action: If operand is 1, signed op = op >> 1 (sign extends op)

If operand is CL, signed op = op >> CL (sign extends op)

Flags Affected: OF, SF, ZF, AF=?, PF, CF

**sbb Subtract with borrow**

Syntax: sbb dest, src

dest: register or memory

src: register, memory, or immediate

Action: dest = dest - (src + CF)

Flags Affected: OF, SF, ZF, AF, PF, CF

Notes: This instruction is used to perform 32-bit subtraction.

**shl Bitwise left shift (same as sal)**

Syntax: shl op, 1

shl op, CL

op: register or memory

Action: If operand is 1, op = op << 1

If operand is CL, op = op << CL

Flags Affected: OF, SF, ZF, AF=?, PF, CF

**shr Bitwise right shift (unsigned)**

Syntax: shr op, 1

shr op, CL

op: register or memory

Action: If operand is 1, op = (unsigned)op >> 1

If operand is CL, op = (unsigned)op >> CL

Flags Affected: OF, SF, ZF, AF=?, PF, CF

**sti Set interrupt flag (enable interrupts)**

Syntax: sti

Action: Set IF

Flags Affected: IF=1

**sub Subtract two numbers**

Syntax: sub dest, src

dest: regsiter or memory

src: register, memory, or immediate

Action: dest = dest - src

Flags Affected: OF, SF, ZF, AF, PF, CF

Notes: Works for both signed and unsigned numbers.

**test Bitwise logical compare**

Syntax: test op1, op2

op1: register, memory, or immediate

op2: register, memory, or immediate

Action: Perform op1 & op2, discarding the result but setting the flags.

Flags Affected: OF=0, SF, ZF, AF=?, PF, CF=0

Notes: This instruction is used to test if bits of a value are set.

**xor Bitwise logical XOR**

Syntax: xor dest, src

dest: register or memory

src: register, memory, or immediate

Action: dest = dest ^ src

Flags Affected: OF=0, SF, ZF, AF=?, PF, CF=0

### C Functions on the 8086 (Near Calls)

When a C function call is made, arguments are passed to the function by being pushed one word (i.e., 16-bits) at a time onto the stack in reverse of the order as listed in the C function declaration. All near functions are of the following form (or an equivalent form):

MyFunc:

push bp ; (1) save bp

mov bp, sp ; (2) set bp for referencing stack

sub sp, {local data size} ; (3) allocate space for local variables

...

... <- function body

...

mov {return reg}, {return value} ; (4) set return value

mov sp, bp ; (5) free space used by local variables

pop bp ; (6) restore bp

ret

The first thing a C function does (1) is save bp. It can then (2) put a copy of sp into bp. Once bp is set up, (3) storage space for any local variables is reserved on the stack. These first three steps set up the *stack frame* for the function. Since the initial value of sp is saved in register bp, bp can be used as a reference to access the arguments that were pushed onto the stack when the function was called as well as the local variables declared within the function. At the end of the function, if the function returns a value, (4) the return value is placed in the appropriate registers. To clean up, (5) the stack is restored to its initial value, freeing the local data space that was allocated in step 3, and (6) bp is restored, thus removing the stack frame. In order for this system to work properly, bp should not be modified in the function body.

**Note 1:** If there are no local variables then there is no need to reserve space on the stack (step (3)). If there are no function arguments or local variables then it may not be necessary to set up and restore the stack frame (steps (1)-(3) and (5)-(6)). If there is no return value for a function then step (4) can be omitted.

**Note 2:** When writing functions in assembly language, the conventions above should always be followed. Also, many compilers (including c86) expect certain registers to be unchanged when a function returns, except for those registers used to save the return value. Therefore, **the stack should be used to save and restore registers that are used in the function body**. For example if I wanted to use the register bx in a function I should execute "push bx" at the beginning of the function and "pop bx" at the end.

Consider the following C function:

int MyFunc(int arg1, int arg2, int arg3)

{

int local1;

int local2;

int local3;

...

... <- function body

...

return 3;

}

For this function, compared to the assembly version above, {local data size} would be 6 for the three word-sized variables (or 6 bytes), {return reg} would be ax because MyFunc returns type int, and {return value} would be 3. Access to the arguments and to the local variables is made relative to bp. For example, the following assembly memory references would be used to access the variables in MyFunc():

[bp+8] -> arg3

[bp+6] -> arg2

[bp+4] -> arg1

[bp+2] -> saved ip (return address)

[bp] -> saved bp

[bp-2] -> local1

[bp-4] -> local2

[bp-6] -> local3

Therefore, if you wanted to load variable local2 into register dx, you could use the following instruction:

mov dx, word [bp-4]

The numbers used in the memory references change based on the size of the data values. However, the first argument is always located at [bp+4] and the first local variable is always just below [bp] (i.e., at [bp-1] for a byte, at [bp-2] for a word, etc.).

**Byte Sized Variables**

Byte sized variables can be a point of confusion. When passing *arguments* to a function, byte sized arguments are always pushed onto the stack as words, since the 8086 push instruction will only push 16-bit values. The *least* significant byte of the word is used to store the value (i.e., the lower memory address). Similarly, for local variables, a full word is reserved for byte sized variables. However, for byte sized *local variables*, the part of the word that is used is the *most* significant byte (i.e., the higher memory address). Using word sizes ensures that there will be no misaligned memory accesses, which can slow performance. It's important to remember when accessing byte sized variables that the unused byte of the word may or may not be zero and should be avoided.

**Local Variable Examples**

The table below shows the effects that changing the above C function has on local variable locations. In the left column is the original example. The middle column shows the effects of changing local1 from type int to type char. The right column shows the effects of changing local1 from type int to type long. The changes are noted in bold.

|  |  |  |
| --- | --- | --- |
| **Original Int Example** | **Char Local Variable** | **Long Local Variable** |
| int MyFunc(...)  {  int local1;  int local2;  int local3;  ...  ...  } | int MyFunc(...)  {  **char** local1;  int local2;  int local3;  ...  ...  } | int MyFunc(...)  {  **long** local1;  int local2;  int local3;  ...  ...  } |
| [bp-2] -> local1 (word)  [bp-4] -> local2 (word)  [bp-6] -> local3 (word) | **[bp-1]** -> local1 (**byte, [bp-2] is unused**)  [bp-4] -> local2 (word)  [bp-6] -> local3 (word) | **[bp-4]** -> local1 (**dword, [bp-2] is high word**)  **[bp-6]** -> local2 (word)  **[bp-8]** -> local3 (word) |

**Argument Examples**

The table below shows the effects that changing the C function example has on argument locations. In the left column is the original example. The middle column shows the effects of changing arg1 from type int to type char. The right column shows the effects of changing arg1 from type int to type long. The changes are noted in bold.

|  |  |  |
| --- | --- | --- |
| **Original Int Example** | **Char Argument** | **Long Argument** |
| int MyFunc(int arg1, int arg2, int arg3)  {  ...  ...  } | int MyFunc(**char** arg1, int arg2, int arg3)  {  ...  ...  } | int MyFunc(**long** arg1, int arg2, int arg3)  {  ...  ...  } |
| [bp+8] -> arg3 (word)  [bp+6] -> arg2 (word)  [bp+4] -> arg1 (word) | [bp+8] -> arg3 (word)  [bp+6] -> arg2 (word)  [bp+4] -> arg1 (**byte, [bp+5] is unused**) | **[bp+10]** -> arg3 (word)  **[bp+8]** -> arg2 (word)  [bp+4] -> arg1 (**dword, [bp+6] is high word**) |

**Return Values**

The following table shows the registers that are used to return values, based on the size of the return type.

byte al

word (2 bytes) ax

dword (4 bytes) dx::ax (i.e., the high word in dx and the low word in ax).

On the 8086 char types are a byte; short, enum, and int types are a word (2 bytes); and long types are a dword (4 bytes). For larger types (e.g., structs), a more sophisticated method is used to return values.

### C Functions on the 8086 (Far Calls)

The following information is supplementary and is not required for any labs.

Far functions are used to make function calls across segment boundaries, which is often required in programs larger than 64 KB. Far calls differ from near calls in that both the CS and IP registers are saved on the stack when the call is made, rather than just saving IP. The only difference between the assembly code of a near function and a far function is that when returning, the far function uses the retf instruction instead of ret. The retf instruction reloads both CS and IP from the stack.

For example, consider the following function:

int far MyFunc(int arg1, int arg2, int arg3)

{

int local1;

int local2;

int local3;

...

...

return 3;

}

Note the far keyword in the declaration. Because an extra word is saved on the stack for the return segment when the function call is made, the arguments placed on the stack are offset by one extra word relative to bp (the local variables remain the same relative to bp). The variables for the above function are located as follows:

[bp+10] -> arg3

[bp+8] -> arg2

[bp+6] -> arg1

[bp+4] -> return CS (segment)

[bp+2] -> return IP (offset)

[bp] -> saved sp

[bp-2] -> local1

[bp-4] -> local2

[bp-6] -> local3

Compare the locations of arg1, arg2, and arg3 to the locations for the near function example. There may also be a different data segment associated with a far function, in which case the DS register is saved and modified at the beginning of the function then restored at the end. This is necessary so that the global and static data associated with the far function can be correctly accessed.

### Quick Reference

**For near calls:**

Location relative to bp | Variable

-----------------------------------------------|------------

... | ...

[bp+4+wsize(arg1)+wsize(arg2)] --------------- | arg3

[bp+4+wsize(arg1)] --------------------------- | arg2

[bp+4] --------------------------------------- | arg1

[bp+2] --------------------------------------- | return address

[bp] ----------------------------------------- | saved bp

[bp-size(local1)] ---------------------------- | local1

[bp-wsize(local1)-size(local2)] -------------- | local2

[bp-wsize(local1)-wsize(local2)]-size(local3)] | local3

... | ...

**For far calls:**

Location relative to bp | Variable

-------------------------------|------------

... | ...

[bp+6+wsize(arg1)+wsize(arg2)] | arg3

[bp+6+wsize(arg1)] ----------- | arg2

[bp+6] ----------------------- | arg1

[bp+4] ----------------------- | return address segment

[bp+2] ----------------------- | return address offset

{same as above} -------------- | local variables

... | ...

**Notes:** size(X) is the size (in bytes) of X and wsize(X) is the word size of X, i.e., the largest multiple of 2 that is at least as big as size(X). The variables local1, local2, etc.  refer to local variables and the variables arg1, arg2, etc. refer to function arguments. The names with a 1 suffix designate those that are declared first in the C source.

Return values are placed in al, ax, or dx:ax for byte, word, and dword sized values respectively.