8086 Instruction set

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

March 11, 2021

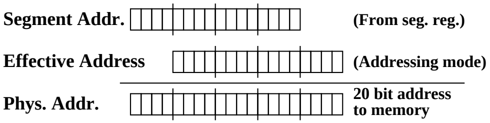
Segmented addresses

A physical memory address is composed of two parts.

- The first part is the segment address, typically held in a segment register in the processor.
- The second part is the offset from this segment address. This is often called the effective address, or EA.
- The physical address is generated by adding the segment address shifted 4 bits to the left to the effective address.
- Thus the 20 bit physical address is given by:
  PA = 16 * Segment address + Effective Address

Segment Registers

The segment address is provided by one of four segment registers in 8086.
The effective address is specified using one of many addressing modes defined for the 8086.



Segment Registers:
CS: Default for code addresses.
DS: Default for data access.
SS: Default for stack addresses.
ES: Extra segment for data.

8086 instruction format permits a segment override field, which allows the use of a segment register other than default.

8086 Register Set

Instruction Pointer

IP

Segment Registers

CS: 16 bit
SS: 16 bit
DS: 16 bit
ES: 16 bit

General Purpose Registers

AH	AL
BH	BL
CH	CL
DH	DL

SP
BP
SI
DI

Covention for specifying address -- seg : offset

“General Purpose” Registers

The register set of an 8086 contains many general purpose 16 bit registers. Some of these permit using each half (8bits) of the register independently.

- Register AX is used as the accumulator, but can be used as a general purpose 16 bit register. Its upper and lower halves (AH and AL) can be used as 8 bit registers independent of each other.
- Similarly, BX, CX and DX permit the use of their 8 bit halves BH-BL, CH-CL and DH-DL independent of each other.
- BX is often used to store the 16 bit Base address of an operand, but can be used as a general purpose 16 bit data register as well.
- CX is often used to store a 16 bit Count for a repetitive operation, but can be used as a general purpose 16 bit data register as well.
- DX is often used as a general purspose Data register. It is also combined with AX for 32 bit operands/result.

“General Purpose” Registers

Some other general purpose registers do not permit independent usage of their 8 bit halves.

- IP is the 16 bit Instruction Pointer, used by default with CS segement register to point to the address from which the next instruction will be fetched.
- SP is the 16 bit Stack Pointer, used by default with SS segment register.
- BP is the base pointer. It is also used by default with SS segment register. SP and BP are used for managing the return address, function arguments and local variables during a function call. It can also be used for general data storage.
- SI and DI are source and destination index registers. These are used as source and destination pointers in string operations used for moving multiple bytes in memory. By default, SI uses DS as the segment register, while DI uses ES.

Addressing modes for Data

Immediate:	The operand can be a byte or a word. The opernad itself is included as a part of the instruction.
Direct:	The operand is in memory. The 16 bit effective address of the operand is included in the instruction. The implied segment register is DS.
Register:	The operand is the content of a named register.

Word operands (16 bit) can be in
AX, BX, CX, DX, SI, DI, SP or BP.

Byte operands (8 bit) can be in
AH, AL, BH, BL, CH, CL, DH or DL.

Addressing modes for Data

Register indirect: The operand is in memory at the address contained in a named register. Only BX, SI or DI can be used for this purpose.

$$EA = (BX)|(DI)|(SI)$$

Register relative: The operand is in memory at the address obtained by adding a constant to the contents of a named register. The constant can be a byte or a word contained in the instruction. The register can be BX, BP, SI or DI.
The instruction arguments specify which register is to be used.

$$EA = (BX)|(BP)|(SI)|(DI) + (\text{sign extended})\text{Byte}|\text{Word}$$

Addressing modes for Data

Based Indexed: The operand is in memory at an address obtained by adding the contents of a base register (BX or BP) and an index register (SI or DI). The registers to be used are specified by the instruction arguments. EA = (BX)|(BP) + (SI)|(DI)

Relative Based Indexed: The operand is in memory. Its address is obtained by adding a byte or word to the contents of a base register and an index register. The registers and the constant are specified by the instruction arguments.

$$EA = (BX)|(BP) + (SI)|(DI) + \text{(sign extended)Byte|Word}$$

Specifying Operand Size

When using a pointer to specify the effective address of an operand in memory, we also need to specify how may bytes starting at this address form the operand.

This may be clear from the other operand of an instruction. For example, MOV AX, [SI] must refer to a word stored at DS:SI, because the destination is AX.

When the size is not clear from the context, we use the keywords: BYTE PTR, WORD PTR and LONG PTR to specify the operand size as one, two or four bytes.

Addressing modes for Code

Addresses of instructions may be specified by several addressing modes:

Intra-segment direct: Being an intra-segment mode, the segment address in CS is not changed.

The instruction carries an offset relative to the current contents of IP. The effective address is generated by adding this offset to IP.

$$EA = IP + \text{(sign extended)Byte|Word}$$

The instruction carries the key word SHORT if a Byte offset is to be used.

The key word NEAR PTR is used if a Word offset is desired.

Addressing modes for Code

Intra-segment indirect: The effective address is specified by any of the data addressing modes described earlier, except the immediate mode.

This mode can only be used in unconditional jumps.

The effective address so specified overwrites the contents of IP.

(It is an absolute value and is not relative to IP).

Since this is an intra segment mode, the segment address is not changed.

Addressing modes for Code

New values have to be specified for IP as well as CS in case of an inter-segment jump. Keyword FAR PTR is used to signal this.

Inter-segment direct: New values for IP as well as for CS are specified as constants contained in the instruction.

Inter-segment indirect: Replaces the contents of IP as well as CS from two consecutive words in memory, whose address is provided by any of the data addressing modes described earlier (except the immediate and register modes).

Data Movement

DATA MOVEMENT INSTRUCTIONS

(No flags are affected.)

MOV	dst, src	(dst) $\leftarrow$ (src)
LEA	reg, src	(reg) $\leftarrow$ EA of src
LDS	reg, src	(reg) $\leftarrow$ (src) (DS) $\leftarrow$ (src + 2)
LES	reg, src	(reg) $\leftarrow$ (src) (ES) $\leftarrow$ (src + 2)
XCHG	op1, op2	(op1)$\leftrightarrow$ (op2)

- Destination cannot be immediate.
- Both operands cannot be in data memory.
- dst cannot be CS.
- For LEA, LDS and LES, the dest reg for EA cannot be a segment register. The source operand cannot be immediate or register.
- For XCHG, neither operand can be a segment register.

PORT input-output

Bytes or Words can be exchanged between the accumulator and specified ports.

IN	AL, Port No.
IN	AL, DX
IN	AX, Port No.
IN	AX, DX
OUT	Port No., AL
OUT	DX, AL
OUT	Port No., AX
OUT	DX, AX

- The src of IN and dst of OUT is a port number.
- If Port No. is < 256, it can be given as an immediate.
- Otherwise, it is placed in DX and DX is given as the argument.

No flags are affected.

Byte Translation

We often need to translate a byte by table look up.
For example, given a key code, we may want to translate it to the character it represents.
The XLAT instruction provides this facility.

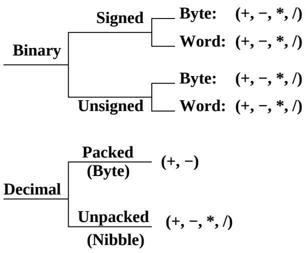
- We set up a table of up to 256 bytes in memory.
- The start address of the table is loaded in BX.
- AL is loaded with the index in this table
- Now the XLAT instruction will replace AL by the element of the table whose index was in AL.

XLAT	Translate Byte in AL	((AL) $\leftarrow$ ((Bx) + (Al))

After XLAT, AL will contain the table entry corresponding to the index.

Arithmetic Instructions

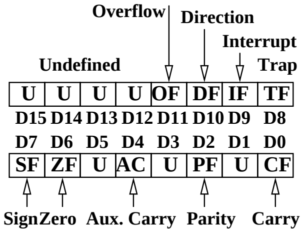
The 8086 can operate on different types and sizes of operands.



Arithmetic operations affect flags.

- Additions, subtractions, multiplications and divisions can be performed on signed and unsigned binary numbers which are 8 bits or 16 bits wide.
- These operations can also be performed on unsigned single decimal digits stored in bytes.
- Unsigned Additions and subtractions can be performed on double decimal digits packed in bytes.

Flags



(Lower Byte is 8085 compatible).

- Overflow: Indicates overflow conditions.
- Direction: Index registers will be incremented (DF=0) or decremented (DF=1) during string operations depending on this flag.
- Interrupt: Enables Interrupts when set
- Trap: Enables single stepping through a program by causing an interrupt (type 3) after executing every instruction.

Binary Addition and Subtraction

ADD	dst, src	(dst)$\leftarrow$ (dst)+(src)
ADC	dst, src	(dst)$\leftarrow$ (dst)+(src)+(CF)
SUB	dst, src	(dst)$\leftarrow$ (dst)-(src)
SBB	dst, src	(dst)$\leftarrow$ (dst)-(src)-(CF)
CMP	op1, op2	(flags) set by (op1)-(op2)
INC	dst	(dst)$\leftarrow$ (dst) + 1
DEC	dst	(dst)$\leftarrow$ (dst) - 1
NEG	dst	(dst)$\leftarrow$ - (dst)

All Flags are affected.

INC and DEC do not set CF.

- dst cannot be immediate.
- For instructions using 2 operands, both cannot be in data memory.
- CMP performs a subtraction, sets flags accordingly and discards the result.

Binary multiplication and division

MUL src	For Byte src: $(AX) \leftarrow (AL) \times (\text{src})$ For Word src: $(DX, AX) \leftarrow (AX) \times (\text{src})$
IMUL src	Same as above: signed
DIV src	For Byte src: $(AX)/(\text{src})$: Quotient $\rightarrow (AL)$ $(AX)/(\text{src})$: Remainder $\rightarrow (AH)$ For Word src: $(DX, AX)/(\text{src})$: Quotient $\rightarrow (AX)$ $(DX, AX)/(\text{src})$: Remainder $\rightarrow (DX)$
IDIV src	Same as above: signed

Binary multiplication and division

- Addressing modes:
  src cannot be immediate.
  All other addressing modes are acceptable.
- Flags:
  MUL and IMUL set CF and OF to 1 if two words/bytes are needed to store the result.
  Otherwise, CF and OF are cleared.
  The remaining condition flags are undefined.
  All condition flags are undefined after DIV and IDIV.
- For signed division, the remainder has the same sign as the dividend.
- Division by 0 causes an interrupt.

Packed BCD arithmetic

Two binary coded decimal digits are packed in a byte.

Decimal adjust can be performed after a binary addition or subtraction.

These instructions operate only on AL.

Let AL=D2D1.

Decimal Adjust after addition/subtraction

DAA	D1 = D1 + 6 if AC = 1 or D1 > 9
	D2 = D2 + 6 if CF = 1 or D2 > 9

DAS	D1 = D1 - 6 if AC = 1 or D1 > 9
	D2 = D2 - 6 if CF = 1 or D2 > 9

Unpacked BCD arithmetic

These instructions assume a single decimal digit per byte. The upper nibble should be 0.
For some operations, the contents of the upper 4 bits are immaterial.
In these cases, we can directly operate on the ASCII code of digits.
These instructions are therefore called ASCII adjust.

AAA	Adjust after addition
AAS	Adjust after subtraction
AAM	Adjust after multiplication
AAD	ASCII Adjust before division

Unpacked BCD arithmetic

AAA	(AL) adjusted after addition, (AH)$\leftarrow$ (AH)+carry from adjustment
AAS	(AL) adjusted after subtraction (AH)$\leftarrow$ (AH)-borrow from adjustment
AAM	(AH)$\leftarrow$ Quotient of (AL)/10 (AL)$\leftarrow$ Remainder of (AL)/10
AAD	(AL)$\leftarrow$ 10*(AH) +(AL) (AH)$\leftarrow$ 0

FLAGS: for AAA and AAS, AC and CF are set if there is an adjustment. The remaining flags are undefined.
AAM and AAD set SF, ZF and PF according to their rules.
OF, AC and CF are undefined.

Flag control

Flag Control

CLC	Clear Carry	(CF)$\leftarrow$0
STC	Set Carry	(CF)$\leftarrow$1
CMC	Complement Carry	$(CF) \leftarrow \overline{(CF)}$
CLD	Clear Direction Flag	(DF)$\leftarrow$0
STD	Set Direction Flag	(DF)$\leftarrow$1
CLI	Clear Interrupt Flag	(IF)$\leftarrow$0
STI	Set Interrupt Flag	(IF)$\leftarrow$1
LAHF	Load AH from Flags	(AH)$\leftarrow$Low Byte of Flags
SAHF	Store AH in Flags	Low Byte of Flags$\leftarrow$(AH)

Logic and Sign Extension

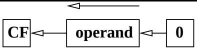
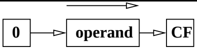
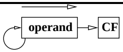
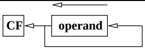
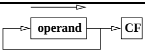
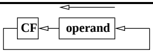
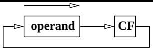
Logic Instructions

NOT op	$(op) \leftarrow \overline{(op)}$
OR dst, src	(dst)← dst) OR (src)
AND dst, src	(dst)← (dst) AND (src)
XOR dst, src	(dst)← (dst) XOR (src)
TEST op1, op2	(Flags)← from (dst) AND (src)

Sign extension

CBW	Convert Byte to Word	Extend sign of AL to AH
CWD	Convert Word to Double	Extend sign of AX to DX

Shift and Rotate

SHL	op, count	
SAL	op, count	Same as SHL
SHR	op, count	
SAR	op, count	
ROL	op, count	
ROR	op, count	
RCL	op, count	
RCR	op, count	

- op can have any addressing mode except immediate.
- If count = 1, it can be specified as immediate.
- If count $>$ 1, the second argument is CL, which contains the number of bits by which we want to shift/rotate.

Branching Instructions

8086 supports several kinds of branching instructions which cause the next instruction to be fetched from an address other than the incremented value of IP.
The instruction carries a key word to show how far the target address of the branch instruction is from the current one.

- SHORT is used for destination within -128 to +127 bytes of updated IP.
- NEAR PTR is used when the destination is within the same segment.
- FAR PTR is used when a new segment address needs to be loaded in CS.

Unconditional Jumps

The instruction JMP can be used in different contexts.
JMP SHORT label
This is a direct, short jump. The assembler calculates the relative offset between the instruction following the jump and the label. This offset should lie between -127 to 127. It is included in the instruction as a signed 8 bit number

JMP NEAR PTR label
This is a direct, near jump. The assembler takes the effective address of the label in the same segment as the instruction following the jump, and puts it as a 16 bit constant in the instruction.

Unconditional Jumps

JMP NEAR PTR TABLE[SI] This is an indirect near jump. The memory is accessed at the specified EA and a word is read from here. This word is put in the IP register. CS remains the same.

JMP FAR PTR label This is a far direct jump. The segment address of this label is put in CS and the offset is put in IP.

JMP FAR PTR TABLE[SI] This is an indirect, far jump. A word is read from the specified address and put in IP. The next word (at address + 2) is also read and put in CS.

Unconditional Jumps

Intra-segment Direct Short: op is a const e.g. a label	
JMP SHORT op	(IP)$\leftarrow$(IP) + sign extended byte op
Intra-segment Direct Near: op is a const e.g. a label	
JMP NEAR PTR op	(IP)$\leftarrow$(IP) + word op
Intra-segment Indirect	
JMP op	(IP)$\leftarrow$(EA) : EA is determined from op
Inter-segment Direct: op is a const e.g. a label	
JMP FAR PTR op	(IP)$\leftarrow$EA of op (CS)$\leftarrow$Seg Addr of op
Inter-segment Indirect	
JMP op	(IP)$\leftarrow$(EA) (CS)$\leftarrow$(EA+2)

Conditional Jumps

All conditional jumps are short. Thus, the target address must be within -128 to +127 bytes from the instruction next to jmp.

If a conditional NEAR or FAR jump is required,
it is constructed by combining a SHORT conditional jump and an unconditional NEAR or FAR jump.

The conditions for jump can be specified as states of single or multiple flags. For example:
JZ label; Jump if ZF is set
JGE label; Jump if SF XOR OF is 0

Conditional Jumps: 8085 like

All these jumps are SHORT and take a signed 8 bit displacement as the argument. In actual use, a label is given and the assembler calculates the displacement.

Mnemonic	Alt. Mnem.	Flag Condition
JZ	JE	ZF=1
JNZ	JNE	ZF=0
JS		SF=1
JNS		SF=0
JP	JPE	PF=1
JNP	JPO	PF=0
JB	JNAE, JC	CF=1
JNB	JAE, JNC	CF=0

Conditional Jumps: Additional

Terms Above and Below are used for unsigned comparison.
Greater and Less Than are used for signed comparison.

Mnemonic	Alt. Mnem.	Flag Condition
JO		OF=1
JNO		OF=0
JBE	JNA	CF+ZF = 1
JNBE	JA	CF+ZF = 0
JL	JNGE	SF $\neq$ OF
JNL	JGE	SF = OF
JLE	JNG	(SF $\neq$ OF) OR ZF = 1
JNLE	JG	(SF = OF) AND ZF = 0

Loop Instructions

Loop instructions decrement CX (without affecting flags), test some condition and then jump conditionally. This combination is useful for looping: hence the name.

Instruction		Alternate Mnemonic	Condition for jump
LOOP	label		CX $\neq$ 0
LOOPZ	label	LOOPE label	ZF=1 & CX $\neq$ 0
LOOPNZ	label	LOOPNE label	ZF=0 & CX $\neq$ 0
JCXZ	label		CX = 0

JCXZ does not decrement CX.
Only SHORT jumps are possible.

CALL

CALLS use the same addressing modes as unconditional jumps, except that a SHORT call is not allowed.
For a direct call, the syntax is:

CALL label

For a NEAR call, the value of updated IP is pushed on the stack, and the offset of label (in code segement) is loaded in IP.

For a FAR call, the value of CS, followed by that of IP is pushed on the stack.
The offset of label is loaded into IP, while its segment address is loaded in CS.

CALL

For an indirect call, the syntax is:

CALL dst

where dst can use any data addressing mode except immediate.

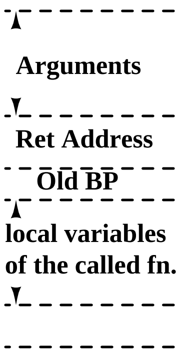
Again, A NEAR call will push only IP,
while a FAR call will push both CS and IP.

Stack Frame for a CALL

Function Call

C | **STACK FRAME** | **ASM**

High level language function call



Calling function pushes function arguments

Assembler Call instruction pushes return address

Entry to called function

PUSH BP; MOV BP, SP

BP points here inside the called fn. ➤

Reserve room for local (auto) variables

Push registers

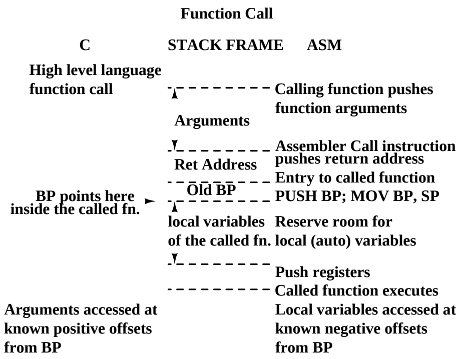
Called function executes

Arguments accessed at known positive offsets from BP

Local variables accessed at known negative offsets from BP

Stack Frame for a CALL

Stack frames are used for managing local variables and function arguments.



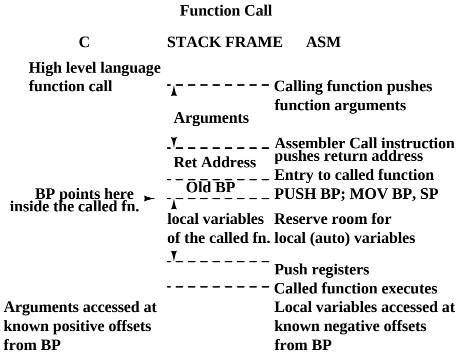
Function Call
C
STACK FRAME
ASM
High level language function call
Calling function pushes function arguments
Arguments
Assembler Call instruction pushes return address
Ret Address
Entry to called function
Old BP
PUSH BP; MOV BP, SP
BP points here inside the called fn.
local variables of the called fn.
Reserve room for local (auto) variables
Push registers
Called function executes
Arguments accessed at known positive offsets from BP
Local variables accessed at known negative offsets from BP

- A high level language function call provides a function name and a list of arguments to the funtions.
- Languages like C support call to a function with a variable number of arguments.
- When a call occurs, the assembler translates it to a series of actions.

Code for the calling function pushes arguments on the stack and then issues an assembly level CALL instruction.

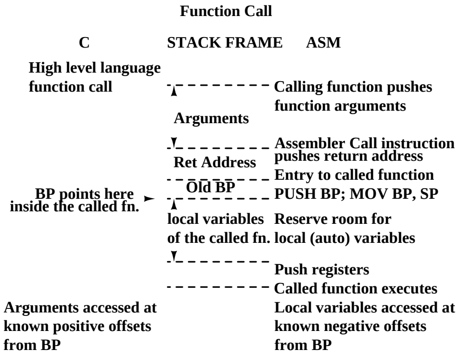
Stack Frame for a CALL

The calling function pushes arguments on the stack and then issues an assembly level CALL instruction to the function to be called.


Function Call
C
STACK FRAME
ASM
High level language function call
Calling function pushes function arguments
Arguments
Assembler Call instruction pushes return address
Ret Address
Entry to called function
Old BP
BP points here inside the called fn.
PUSH BP; MOV BP, SP
local variables of the called fn.
Reserve room for local (auto) variables
Push registers
Called function executes
Arguments accessed at known positive offsets from BP
Local variables accessed at known negative offsets from BP

- This places the return address on the stack after the arguments.
- The offset of the code for the called function is loaded in IP, so the next instruction executed is the first instruction of the called function.

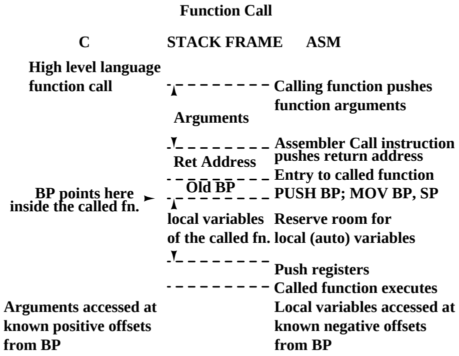
Stack Frame for a CALL


Function Call
C
STACK FRAME
ASM
High level language function call
Calling function pushes function arguments
Arguments
Assembler Call instruction pushes return address
Ret Address
Entry to called function
Old BP
PUSH BP; MOV BP, SP
BP points here inside the called fn.
local variables of the called fn.
Reserve room for local (auto) variables
Push registers
Called function executes
Arguments accessed at known positive offsets from BP
Local variables accessed at known negative offsets from BP

- The called function pushes the base pointer register on the stack to save its value. (It will use BP to locate function arguments and local variables).
- It now copies the current value of the stack pointer to BP.
- BP will now be a stable reference value for locating arguments etc., whereas the value of SP will keep changing as we save registers etc.

The called program now subtracts the size of storage required for local variables from SP. (This size is known to the called function).

Stack Frame for a CALL



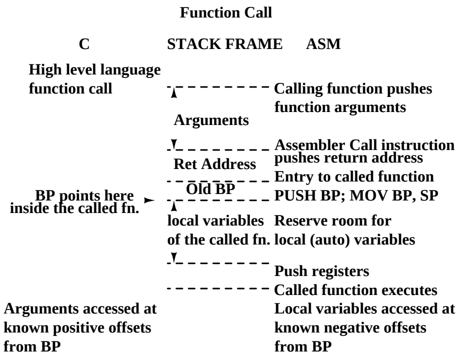
Function Call
C
STACK FRAME
ASM
High level language function call
Calling function pushes function arguments
Arguments
Assembler Call instruction pushes return address
Ret Address
Entry to called function
Old BP
PUSH BP; MOV BP, SP
BP points here inside the called fn.
local variables of the called fn.
Reserve room for local (auto) variables
Push registers
Called function executes
Arguments accessed at known positive offsets from BP
Local variables accessed at known negative offsets from BP

- The called function saves registers on the stack and starts its execution.
- It can access the arguments to the function at known (and fixed) positive offsets from BP.
- It can access local variables at known (and fixed) negative offsets from BP.

The entire data structure containing arguments, return address, saved BP and local variables is called a stack frame.

Return from a high level language function call

The return from a function called in a high level language reverses the actions which were used to construct the stack frame.

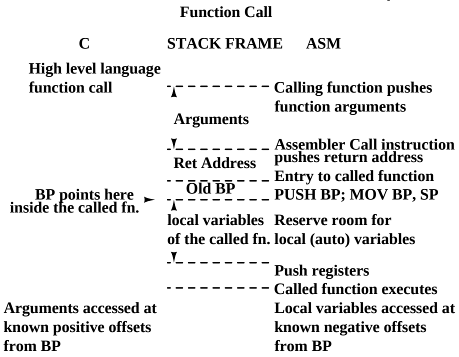

Function Call
C
STACK FRAME
ASM
High level language function call
Calling function pushes function arguments
Arguments
Assembler Call instruction pushes return address
Ret Address
Entry to called function
Old BP
PUSH BP; MOV BP, SP
BP points here inside the called fn.
local variables of the called fn.
Reserve room for local (auto) variables
Push registers
Called function executes
Arguments accessed at known positive offsets from BP
Local variables accessed at known negative offsets from BP

- When the called function has finished its job, it pops the registers it had pushed on stack (in reverse order).
- After that, it moves BP back to SP. This frees up the space occupied by the local variables.
- Now it pops BP from the stack to restore its original value.

An assembly level RET instruction will now pop the return addrss from the stack and put it in IP. The calling function should now remove the arguments from the stack.

Return from a high level language function call

After the saved data registers, local variables and the saved value of BP have been removed from the stack frame, the return address is at the top of the stack.


Function Call
C
STACK FRAME
ASM
High level language function call
Calling function pushes function arguments
Arguments
Assembler Call instruction pushes return address
Ret Address
Entry to called function
Old BP
PUSH BP; MOV BP, SP
BP points here inside the called fn.
local variables of the called fn.
Reserve room for local (auto) variables
Push registers
Called function executes
Arguments accessed at known positive offsets from BP
Local variables accessed at known negative offsets from BP

- An assembly level return instruction will pop the return address from the stack and copy it to IP (or to CS:IP if the call was to a "far" function).
- The next instruction to be executed is the instruction next to the call instruction in the calling program.

The calling program knows the size of of the arguments it has pushed on the stack. It adds this size to SP to restore it to the value it had before the high level function call occurred.

RET assembly instruction

The RET instruction pops the return address from the stack.

It can (optionally) adjust the value of the stack pointer to account for the parameters placed on the stack before calling a subroutine.

A NEAR return pops just the IP value.
A FAR return pops both IP and CS values.

The “model” used for compiling the program determines whether a NEAR or FAR CALL and RET will be used.

RETURN

RET expression

The expression is optional and must evaluate to a constant at compile time.

If used, it places a 16bit number in the instruction, which is added to SP after the return address has been popped.

This effectively removes any arguments for the called function that might have been placed on the stack.

8086 Program models

Program models determine if references to data and code will be NEAR (segment registers retain their values through the program) or FAR (segment as well as offset values must be manipulated) by default.

SMALL model: Here Code and data size are restricted to 64 KB each. Code as well as data pointers just use the offset value. Segement registers are initialized at the start and need not be adjusted afterwards.

COMPACT model: Here the code segment is smaller that 64 KB, though data segment exceeds this limit. Here code pointers are 16 bit wide, and code segment register is initialized once and for all.
Data references have to specify segment as well as offset values for each data item.

8086 Program models

MEDIUM model: Here data size is restricted to 64 KB, but code can be larger than 64 KB.
Data Segement registers are initialized at the start and need not be adjusted afterwards.
However, code pointers for jump, call, return etc. need to specify the values for code segement as the offset in that segment.

LARGE model: Here the code as well as data segments are larger than 64 KB. Thus segment as well as offset values have to be specified for both data and code pointers.

In addition to these four models, the TINY model can be used when the total code + data size is less than 64 KB.

The HUGE model has to be used when a single data object like an array can exceed 64 KB.

INTERRUPTS

- When an interrupt occurs (in hardware), an interrupt number is read from the interrupting peripheral.
- A software interrupts specifies the interrupt no. explicitly or implicitly.
- The full 16 bit flag register (PSW) is pushed on the stack.
- Then CS and updated IP are pushed onto the stack.
- 1K byte of memory at 0000:0000 is treated like a table of FAR addresses.
- The interrupt no. is used as an index in this table, the WORD (at 4*index) is loaded into IP, and the next one (at 4*index+2) into CS.
- The interrupt service routine located at this CS:IP must return using IRET.

Interrupts and IRET

Instruction	Description
INT TYPE	PUSH PSW, PUSH CS, PUSH IP (IP)←(4*TYPE) (CP)←(4*TYPE+2)
INT	Same: default TYPE = 3

INTO	IF OF=1,
	same as above with TYPE = 4,
	else NOP

IRET	POP IP, POP CS, POP PSW

String instructions

- These are efficient instructions for performing repeated operations on a region in memory.
- The basic instructions perform a single operation, but with a REP prefix, a single string instruction will auto-repeat till some condition is met.

String instructions

- There are 5 string instructions: MOVS, CMPS, SCAS, LODS and STOS.
- Each one may be used in one of the three forms:
    - OperationS Operand(s)
    - OperationB
    - OperationW
- The operands are actually implied, but may be explicitly specified for self documentation and for fixing the size of operations.
- Suffixes B and W specify the size of the operation as Byte or Word without naming the (implied) operands.

Move String

Moves data from DS:SI to ES:DI.

Instruction	Description
MOVS dst, src	((DI))←((SI))
MOVSB	Byte ops: (SI) = (SI) ±1 (DI) = (DI) ±1
MOVSW	Word ops: (SI) = (SI) ±2 (DI) = (DI) ±2

src is DS:SI by default - but the source segment register may be overridden by ES, SS or CS.

dst is always ES:DI.

DF = 0 increments, DF = 1 decrements SI and DI.

Compare String

Compares data at DS:SI with that at ES:DI

Instruction	Description
CMPS dst, src	Set flags based on ((SI)) - ((DI))
CMPSB	Byte ops: (SI) = (SI) ± 1 (DI) = (DI) ± 1
CMPSW	Word ops: (SI) = (SI) ± 2 (DI) = (DI) ± 2

src is DS:SI by default - but the source segment register may be overridden by ES, SS or CS.

dst is always ES:DI.

DF = 0 increments, DF = 1 decrements SI and DI.

Scan String

Compares AL/AX with data at ES:DI

Instruction	Description
SCAS dst	For Byte operands:
	Set flags based on
	(AL) - ((DI))
SCASB	Then
	(DI) = (DI) $\pm$1
SCASW	Word operands:
	Set flags based on
	(AX) - ((DI))
	Then
	(DI) = (DI) $\pm$2

Load/Store string

Move data between AL/AX and memory, with auto-adjustment of pointer values

Instruction	Description
LODS src	Byte operands:
LODSB	(AL)$\leftarrow$((SI))
	(SI) = (SI)$\pm$1
LODSW	Word operands:
	(AX)$\leftarrow$((SI))
	(SI) = (SI)$\pm$2

Instruction	Description
STOS src	Byte operands:
STOSB	((DI))$\leftarrow$(AL)
	(DI) = (DI)$\pm$1
STOSW	Word operands:
	((DI))$\leftarrow$(AX)
	(DI) = (DI)$\pm$2

String Instructions with REP

String instructions can be auto repeated a number of times by loading a count in CX and using the REP prefix.
As in the case of a loop instruction,
CX is decremented in every iteration
and looping continues till some conditions are met:

Prefix	Alt. Name	Termination Condition
REP		CX = 0
REPZ	REPE	ZF=0 OR CX=0
REPNZ	REPNE	ZF=1 OR CX=0

REP prefix

The REP prefix works as follows:

1. If CX=0 (or ZF is clear/set in case of REPZ/REPNZ), exit.
2. Perform the string primitive.
3. Decrement CX by 1 without changing flags.
4. Repeat the above 3 steps till the termination condition is met.