Arithmetic and Logic Instructions

Introduction

- We examine the arithmetic and logic instructions. The arithmetic instructions include addition, subtraction, multiplication, division, comparison, negation, increment, and decrement.
- The logic instructions include AND, OR, Exclusive-OR, NOT, shifts, rotates, and the logical compare (TEST).

ADD

- The general format of ADD instruction is
- ADD destination, source
- The ADD instruction directs the CPU to add the value contained in the destination operand and put the result in destination operand

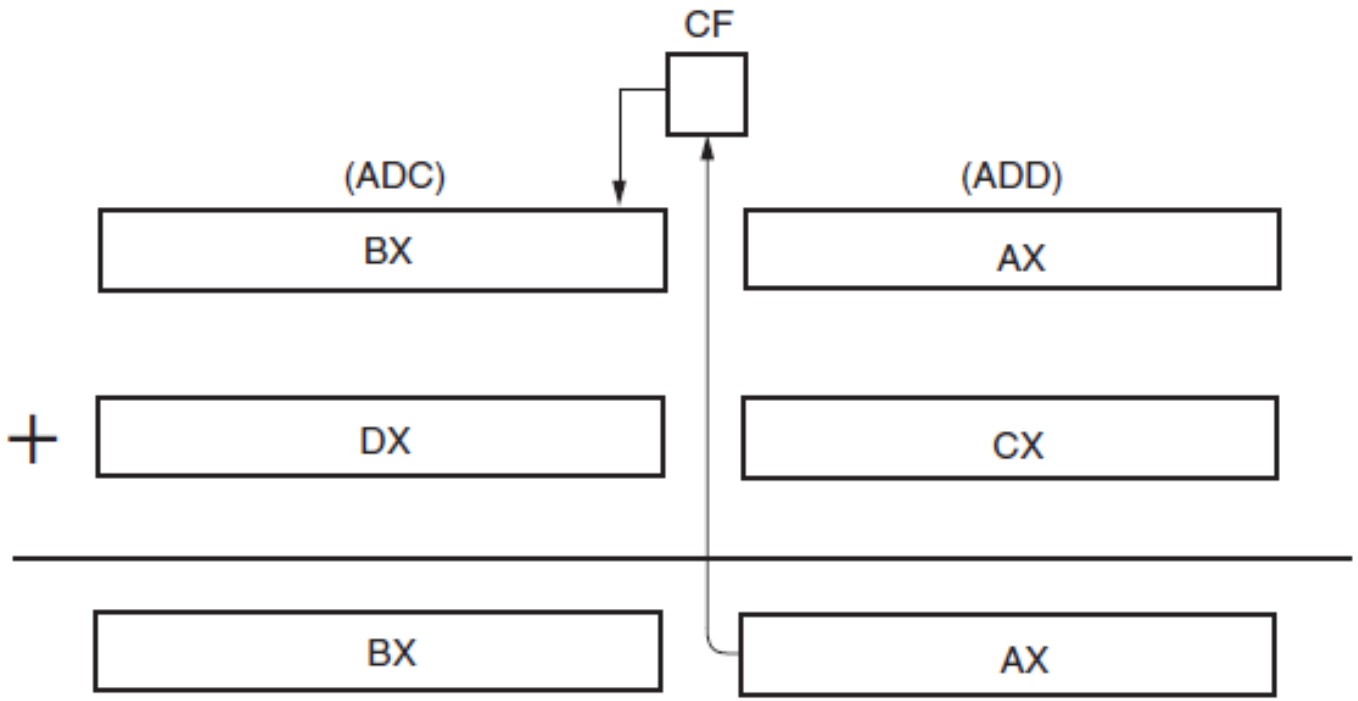
Register, Memory	(e.g., ADD AX, Var1)
Memory, Register	(e.g., ADD Var1, AX)
Register, Register	(e.g., ADD AX, BX)
Memory, Immediate	(e.g., ADD Var1, 4)
Register, Immediate	(e.g., ADD AX, 4)

Assembly Language	*Operation*
ADD AL,BL	AL = AL + BL
ADD CX,DI	CX = CX + DI
ADD EBP,EAX	EBP = EBP + EAX
ADD CL,44H	CL = CL + 44H
ADD BX,245FH	BX = BX + 245FH
ADD EDX,12345H	EDX = EDX + 12345H
ADD [BX],AL	AL adds to the byte contents of the data segment memory location addressed by BX with the sum stored in the same memory location
ADD CL,[BP]	The byte contents of the stack segment memory location addressed by BP add to CL with the sum stored in CL
ADD AL,[EBX]	The byte contents of the data segment memory location addressed by EBX add to AL with the sum stored in AL
ADD BX,[SI+2]	The word contents of the data segment memory location addressed by SI + 2 add to BX with the sum stored in BX
ADD CL,TEMP	The byte contents of data segment memory location TEMP add to CL with the sum stored in CL
ADD BX,TEMP[DI]	The word contents of the data segment memory location addressed by TEMP + DI add to BX with the sum stored in BX
ADD [BX+D],DL	DL adds to the byte contents of the data segment memory location addressed by BX + DI with the sum stored in the same memory location
ADD BYTE PTR [DI],3	A 3 adds to the byte contents of the data segment memory location addressed by DI with the sum stored in the same location
ADD BX,[EAX+2*ECX]	The word contents of the data segment memory location addressed by EAX plus 2 times ECX add to BX with the sum stored in BX
ADD RAX,RBX	RBX adds to RAX with the sum stored in RAX (64-bit mode)
ADD EDX,[RAX+RCX]	The doubleword in EDX is added to the doubleword addressed by the sum of RAX and RCX and the sum is stored in EDX (64-bit mode)

Addition-with-Carry

- An addition-with-carry instruction (ADC) adds the bit in the carry flag (C) to the operand data.
- ADC affects the flags after the addition

Assembly Language	*Operation*
ADC AL,AH	AL = AL + AH + carry
ADC CX,BX	CX = CX + BX + carry
ADC EBX,EDX	EBX = EBX + EDX + carry
ADC RBX,0	RBX = RBX + 0 + carry (64-bit mode)
ADC DH,[BX]	The byte contents of the data segment memory location addressed by BX add to DH with the sum stored in DH
ADC BX,[BP+2]	The word contents of the stack segment memory location addressed by BP plus 2 add to BX with the sum stored in BX
ADC ECX,[EBX]	The doubleword contents of the data segment memory location addressed by EBX add to ECX with the sum stored in ECX



Addition with-carry showing how the carry flag (C) links the two 16-bit additions into one 32-bit addition.

Increment Addition

- Increment addition (INC) adds 1 to a register or a memory location.
- INC instruction adds 1 to any register or memory location, except a segment register

Assembly Language	*Operation*
INC BL	BL = BL + 1
INC SP	SP = SP + 1
INC EAX	EAX = EAX + 1
INC BYTE PTR[BX]	Adds 1 to the byte contents of the data segment memory location addressed by BX
INC WORD PTR[SI]	Adds 1 to the word contents of the data segment memory location addressed by SI
INC DWORD PTR[ECX]	Adds 1 to the doubleword contents of the data segment memory location addressed by ECX
INC DATA1	Adds 1 to the contents of data segment memory location DATA1
INC RCX	Adds 1 to RCX (64-bit mode)

Subtraction

- The general format of the SUB instruction is
- SUB destination, source
- SUB instruction directs the CPU to subtract the value contained in the source operand from the value contained in the destination operand and put the result in the destination operand.

Register, Memory	(e.g., SUB AX, Var1)
Memory, Register	(e.g., SUB Var1, AX)
Register, Register	(e.g., SUB AX, BX)
Memory, Immediate	(e.g., SUB Var1, 4)
Register, Immediate	(e.g., SUB AX, 4)

Immediate Subtraction

- microprocessor also allows immediate operands for the subtraction of constant data

```
MOV CH,22H
SUB CH,44H
```

Z = 0 (result not zero)
C = 1 (borrow)
A = 1 (half-borrow)
S = 1 (result negative)
P = 1 (even parity)
O = 0 (no overflow)

Flags changed

Assembly Language	*Operation*
SUB CL,BL	CL = CL – BL
SUB AX,SP	AX = AX – SP
SUB ECX,EBP	ECX = ECX – EBP
SUB RDX,R8	RDX = RDX – R8 (64-bit mode)
SUB DH,6FH	DH = DH – 6FH
SUB AX,0CCCCH	AX = AX – 0CCCCH
SUB ESI,2000300H	ESI = ESI – 2000300H
SUB [DI],CH	Subtracts CH from the byte contents of the data segment memory addressed by DI and stores the difference in the same memory location
SUB CH,[BP]	Subtracts the byte contents of the stack segment memory location addressed by BP from CH and stores the difference in CH
SUB AH,TEMP	Subtracts the byte contents of memory location TEMP from AH and stores the difference in AH
SUB DI,TEMP[ESI]	Subtracts the word contents of the data segment memory location addressed by TEMP plus ESI from DI and stores the difference in DI
SUB ECX,DATA1	Subtracts the doubleword contents of memory location DATA1 from ECX and stores the difference in ECX
SUB RCX,16	RCX = RCX – 18 (64-bit mode)

Decrement Subtraction

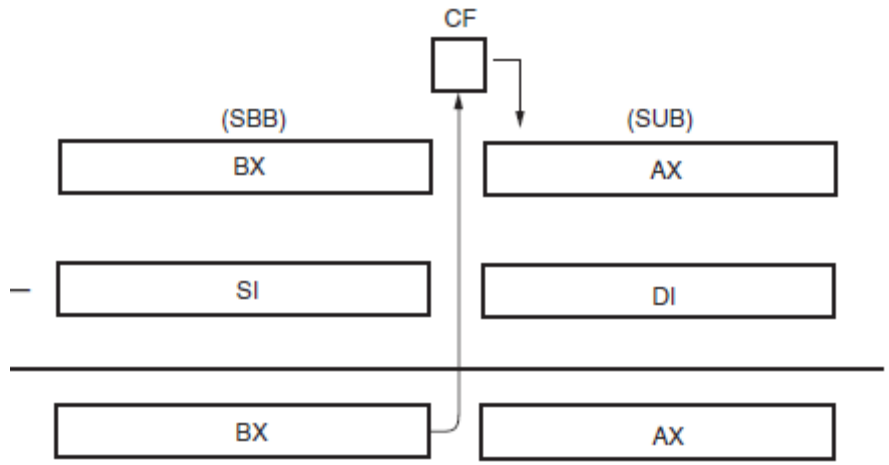
- Decrement subtraction (DEC) subtracts 1 from a register or the contents of a memory location.

Assembly Language	*Operation*
DEC BH	BH = BH – 1
DEC CX	CX = CX – 1
DEC EDX	EDX = EDX – 1
DEC R14	R14 = R14 – 1 (64-bit mode)
DEC BYTE PTR[DI]	Subtracts 1 from the byte contents of the data segment memory location addressed by DI
DEC WORD PTR[BP]	Subtracts 1 from the word contents of the stack segment memory location addressed by BP
DEC DWORD PTR[EBX]	Subtracts 1 from the doubleword contents of the data segment memory location addressed by EBX
DEC QWORD PTR[RSI]	Subtracts 1 from the quadword contents of the memory location addressed by RSI (64-bit mode)
DEC NUMB	Subtracts 1 from the contents of data segment memory location NUMB

Subtraction-with-Borrow

- A subtraction-with-borrow (SBB) instruction functions as a regular subtraction, except that the carry flag (C), which holds the borrow, also subtracts from the difference.

Assembly Language	*Operation*
SBB AH,AL	AH = AH – AL – carry
SBB AX,BX	AX = AX – BX – carry
SBB EAX,ECX	EAX = EAX – ECX – carry
SBB CL,2	CL = CL – 2 – carry
SBB RBP,8	RBP = RBP– 2 – carry (64-bit mode)
SBB BYTE PTR[DI],3	Both 3 and carry subtract from the data segment memory location addressed by DI
SBB [DI],AL	Both AL and carry subtract from the data segment memory location addressed by DI
SBB DI,[BP+2]	Both carry and the word contents of the stack segment memory location addressed by BP plus 2 subtract from DI
SBB AL,[EBX+ECX]	Both carry and the byte contents of the data segment memory location addressed by EBX plus ECX subtract from AL


CF
(SBB)
(SUB)
BX
AX
SI
DI
BX
AX

Comparison

- The comparison instruction (CMP) is a subtraction that changes only the flag bits;
- The destination operand never changes. A comparison is useful for checking the entire contents of a register or a memory location against another value.
- A CMP is normally followed by a conditional jump instruction, which tests the condition of the flag bits.

```
CMP AL,10H
JAE SUBER
```

Assembly Language	*Operation*
CMP CL,BL	CL – BL
CMP AX,SP	AX – SP
CMP EBP,ESI	EBP – ESI
CMP RDI,RSI	RDI – RSI (64-bit mode)
CMP AX,2000H	AX – 2000H
CMP R10W,12H	R10 (word portion) – 12H (64-bit mode)
CMP [DI],CH	CH subtracts from the byte contents of the data segment memory location addressed by DI
CMP CL,[BP]	The byte contents of the stack segment memory location addressed by BP subtracts from CL
CMP AH,TEMP	The byte contents of data segment memory location TEMP subtracts from AH
CMP DI,TEMP[BX]	The word contents of the data segment memory location addressed by TEMP plus BX subtracts from DI
CMP AL,[EDI+ESI]	The byte contents of the data segment memory location addressed by EDI plus ESI subtracts from AL

Multiplication

- The format of MUL is
- MUL operand

Register	(e.g., MUL BL)
Memory	(e.g., MUL Var1)

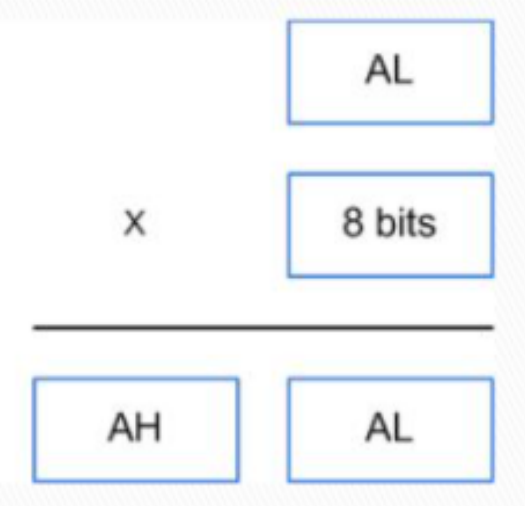
- The operation is as follows:

✓If the operation is a MUL instruction is 16 bit wide, then it is the multiplier and the AX register is the multiplicand

✓The product appears in the DX/AX register pair

16 bit multiplication

- ✓ If the operand of a MUL instruction is only 8bits wide, then it is the multiplier and the AL register is the multiplicand. The product appears in the AX register.



8 bits multiplication

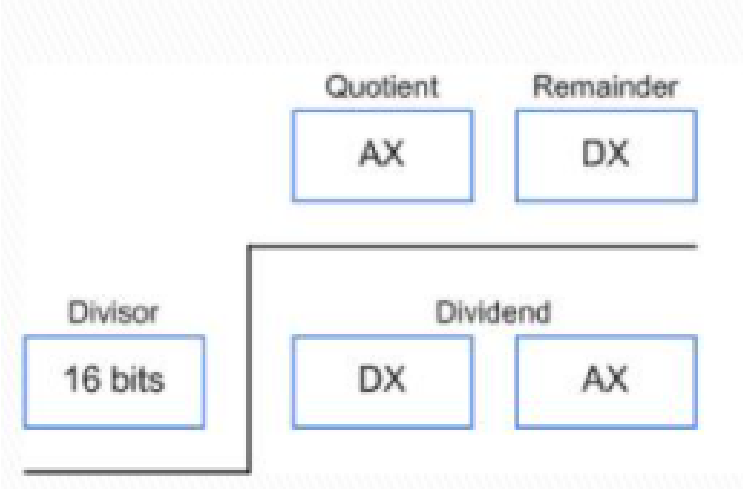
Assembly Language	*Operation*
MUL CL	AL is multiplied by CL; the unsigned product is in AX
IMUL DH	AL is multiplied by DH; the signed product is in AX
IMUL BYTE PTR[BX]	AL is multiplied by the byte contents of the data segment memory location addressed by BX; the signed product is in AX
MUL TEMP	AL is multiplied by the byte contents of data segment memory location TEMP; the unsigned product is in AX

Division

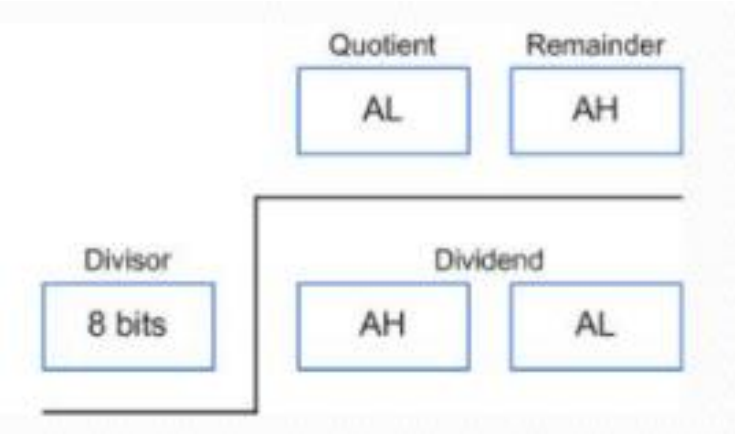
- The format of division instruction is
- DIV operand

Division operation is as follows:

- If the operand in a DIV instruction is 16bits wide, the contents of the operand are divided into the DX/AX register pair
- Integer part of the quotient is stored in AX, and the remainder is stored in DX



- If the operand of a DIV instruction is only 8bits wide, the contents of the operand are divided into AX.
- Integer part of the quotient is stored in AL, and the remainder is stored in AH.



BCD and ASCII Arithmetic

- The microprocessor allows arithmetic manipulation of both BCD (**binary-coded decimal**) and ASCII (**American Standard Code for Information Interchange**) data.
- BCD operations occur in systems such as point-of-sales terminals (e.g., cash registers) and others that seldom require complex arithmetic.

BCD Arithmetic

- Two arithmetic techniques operate with BCD data: addition and subtraction.
- DAA (**decimal adjust after addition**) instruction follows BCD addition,
- DAS (**decimal adjust after subtraction**) follows BCD subtraction.
  - both correct the result of addition or subtraction so it is a BCD number

DAA Instruction

- DAA follows the ADD or ADC instruction to adjust the result into a BCD result.
- After adding the BL and DL registers, the result is adjusted with a DAA instruction before being stored in CL.

DAS Instruction

- Functions as does DAA instruction, except it follows a subtraction instead of an addition.

EXAMPLE 5–18

```
0000 BA 1234        MOV  DX,1234H       ;load 1234 BCD
0003 BB 3099        MOV  BX,3099H       ;load 3099 BCD
0006 8A C3          MOV  AL,BL          ;sum BL and DL
0008 02 C2          ADD  AL,DL
000A 27             DAA
000B 8A C8          MOV  CL,AL          ;answer to CL
000D 9A C7          MOV  AL,BH          ;sum BH, DH an carry
000F 12 C6          ADC  AL,DH
0011 27             DAA
0012 8A E8          MOV  CH,AL          ;answer to CH
```

```
MOV  AL,71H
SUB  AL,43H    ;   AL = 2EH
DAS            ;   AL = 28 H
```

- If the least significant four bits in AL are>9 or if AF=1, it subtracts 6 from AL and sets AF
- If the most significant four bits in AL are >9 or if CF=1, it subtracts 60 from AL and sets the CF

Processing Packed BCD Numbers

- Two instructions to process packed BCD numbers

1. DAA – Decimal Adjust after addition used after ADD or ADC instruction
2. DAS – Decimal adjust after subtraction used after SUB or SBB instruction

- No support for multiplication or division

ASCII Arithmetic

- ASCII arithmetic instructions function with coded numbers, value 30H to 39H for 0–9.
- Four instructions in ASCII arithmetic operations:
  - AAA (**ASCII adjust after addition**)
  - AAD (**ASCII adjust before division**)
  - AAM (**ASCII adjust after multiplication**)
  - AAS (**ASCII adjust after subtraction**)
- These instructions use register AX as the source and as the destination.

BCD Number System

- **BCD number system:**
- In computer literature one encounters two terms for BCD numbers:
- **Unpacked BCD**
- In unpacked BCD, the lower 4 bits of the number represent the BCD number and the rest of the bits are 0.
- Example: 0000 1001 and 0000 0101 are unpacked BCD for 9 and 5, respectively.
- **Packed BCD**
- In the case of packed BCD, a single byte has two BCD numbers in it, one in the lower 4 bits and one in the upper 4 bits.
- Example: 0101 1001 is packed BCD for 59. It takes only a byte of memory to store the packed BCD operands.

ASCII numbers

Key	ASCII (hex)	Binary	BCD unpacked
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

ASCII to BCD conversion:

ASCII to unpacked BCD conversion:

To convert ASCII data to BCD, the programmer must get rid of the tagged "011" in the highest 4 bits of the ASCII. To do that, each ASCII number is ANDed with "0000 1111" (0FH).

ASCII to packed BCD conversion:

To convert ASCII to packet BCD, it is first converted to unpacked BCD (to get rid of 3) and then combined to make packed BCD. For example, for 9 and 5 the keyboard gives 39 and 35 receptively. The goal is to produce 95H or "1001 0101", which is called packed BCD.

Key	ASCII	Unpacked BCD	Packed BCD
4	34	0000 0100	0100 0111
7	37	0000 0111	Or 47H

AAA Instruction

- ASCII adjust AL after addition.
- Adjusts the sum of two unpacked BCD values to create an unpacked BCD result.
- The AL register is the implied source and destination operand for this instruction.
- only useful when it follows an ADD instruction

```
MOV AL, ‘5’    ; AL=35
ADD AL, ‘2’    ; add to AL 32 the ASCII of 2
AAA            ; changes 67H to 07H
OR AL, 30      ; OR AL with 30 to get ASCII
```

AAS Instruction

- Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result.
- The AL register is the implied source and destination operand for this instruction.
- only useful when it follows an SUB instruction.

```
MOV AH, 0          ; AH=0
MOV AL, ‘8’        ;AX=0038H
SUB AL, ‘9’        ;AX=00FFH
AAS                 ;AX=FF09H
OR AL, 30h          ;AX=FF39H
```

AAM Instruction

- Adjust the result of the multiplication of two unpacked BCD values

to create a pair of unpacked BCD values.

- The AX register is the implied source and destination.
- The AAM instruction is only useful when it follows a MUL instruction.

```
MOV BL, 5
MOV AL, 6
MUL BL      ; AX= 001EH
AAM         ; AX= 0300H
```

- The AAM instruction is used to adjust the content of the AL and AH registers after the AL register has been used to perform the multiplication of two unpacked BCD bytes. The CPU uses the following simple logic: al = al mod 10
  ah = al/10

AAD Instruction

- Appears before a division.
- The AAD instruction requires the AX register contain a two-digit unpacked BCD number (not ASCII) before executing.
- Before dividing the unpacked BCD by another unpacked BCD, AAD is used to convert it to HEX. By doing that the quotient and reminder are both in unpacked BCD.

```
MOV AX,3539H ;AX=3539 ASCII for 59
AND AX,0F0FH ; AH=05, AL=09 unpacked BCD data
AAD          ; AX=003BH hex equivalent of 59
MOV BH,08H   ; divide by 08
DIV BH       ; 3B/08 gives AL=07, AH=03
OR  AX,3030H ; AL=37H (quotient) AH=33H (rem)
```

LOGICAL INSTRUCTIONS

AND

- Performs logical multiplication, illustrated by a truth table
- AND op1, op2
- This performs a bitwise Logical AND of two operands.
- The result of the operation is stored in the op1 and used to set the flags.
- Each bit of the result is set to 1 if the corresponding bit in both of the operands are 1. Otherwise the bit in the result is cleared to 0

A	B	T
0	0	0
0	1	0
1	0	0
1	1	1

- AND clears bits of a binary number called as masking

```
x x x x  x x x x   Unknown number
0 0 0 0  1 1 1 1   Mask
---------------
0 0 0 0  x x x x   Result
```

- AND uses any mode except memory-to memory and segment register addressing.
- An ASCII number can be converted to BCD by using AND to mask off the leftmost four binary bit positions

Assembly Language	*Operation*
AND AL,BL	AL = AL and BL
AND CX,DX	CX = CX and DX
AND ECX,EDI	ECX = ECX and EDI
AND RDX,RBP	RDX = RDX and RBP (64-bit mode)
AND CL,33H	CL = CL and 33H
AND DI,4FFFH	DI = DI and 4FFFH
AND ESI,34H	ESI = ESI and 34H
AND RAX,1	RAX = RAX and 1 (64-bit mode)
AND AX,[DI]	The word contents of the data segment memory location addressed by DI are ANDed with AX
AND ARRAY[SI],AL	The byte contents of the data segment memory location addressed by ARRAY plus SI are ANDed with AL
AND [EAX],CL	CL is ANDed with the byte contents of the data segment memory location addressed by ECX

OR

- Performs logical addition
- The OR instruction uses any addressing mode except segment register addressing
- The OR function generates a logic 1 output if any inputs are 1.
- The OR gate sets (1) for any bit of a binary number.

```
  x x x x  x x x x   Unknown number
+ 0 0 0 0  1 1 1 1   Mask
-------------------
  x x x x  1 1 1 1   Result
```

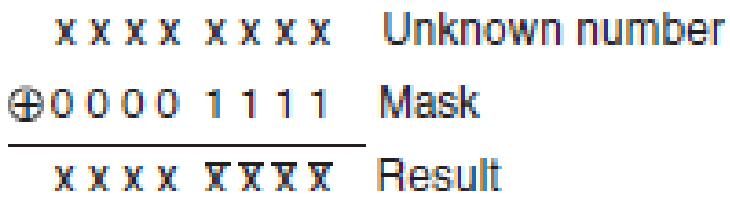
A	B	T
0	0	0
0	1	1
1	0	1
1	1	1

Assembly Language	*Operation*
OR AH,BL	AL = AL or BL
OR SI,DX	SI = SI or DX
OR EAX,EBX	EAX = EAX or EBX
OR R9,R10	R9 = R9 or R10 (64-bit mode)
OR DH,0A3H	DH = DH or 0A3H
OR SP,990DH	SP = SP or 990DH
OR EBP,10	EBP = EBP or 10
OR RBP,1000H	RBP = RBP or 1000H (64-bit mode)
OR DX,[BX]	DX is ORed with the word contents of data segment memory location addressed by BX
OR DATES[DI + 2],AL	The byte contents of the data segment memory location addressed by DI plus 2 are ORed with AL

Exclusive-OR

- Differs from Inclusive-OR (OR) in that the 1,1 condition of Exclusive-OR produces a 0.
- a 1,1 condition of the OR function produces a 1
- The Exclusive-OR operation excludes this condition; the Inclusive-OR includes it.
- If inputs of the Exclusive-OR function are both 0 or both 1, the output is 0; if the inputs are different, the output is 1
- Exclusive-OR is sometimes called a comparator
- XOR uses any addressing mode except segment register addressing.

- Exclusive-OR is useful if some bits of a register or memory location must be inverted
- Figure shows how just part of an unknown quantity can be inverted by XOR.
  - when a 1 Exclusive-ORs with X, the result is X’
  - if a 0 Exclusive-ORs with X, the result is X



Assembly Language	*Operation*
XOR CH,DL	CH = CH xor DL
XOR SI,BX	SI = SI xor BX
XOR EBX,EDI	EBX = EBX xor EDI
XOR RAX,RBX	RAX = RAX xor RBX (64-bit mode)
XOR AH,0EEH	AH = AH xor 0EEH
XOR DI,00DDH	DI = DI xor 00DDH
XOR ESI,100	ESI = ESI xor 100
XOR R12,20	R12 = R12 xor 20 (64-bit mode)
XOR DX,[SI]	DX is Exclusive-ORed with the word contents of the data segment memory location addressed by SI
XOR DEAL[BP+2],AH	AH is Exclusive-ORed with the byte contents of the stack segment memory location addressed by BP plus 2

Test and Bit Test Instructions

- TEST performs the AND operation
  - only affects the condition of the flag register, which indicates the result of the test
  - functions the same manner as a CMP
  - Normally tests a single bit or multiple bits
- Usually followed by either the JZ (jump if zero) or JNZ (jump if not zero) instruction.
  - Z=0 if the bit under test is not zero
  - Z=1 if the bit under test is a zero

- The destination operand is normally tested against immediate data (indicating the bit weight).

Assembly Language	*Operation*
TEST DL,DH	DL is ANDed with DH
TEST CX,BX	CX is ANDed with BX
TEST EDX,ECX	EDX is ANDed with ECX
TEST RDX,R15	RDX is ANDed with R15 (64-bit mode)
TEST AH,4	AH is ANDed with 4
TEST EAX,256	EAX is ANDed with 256

NOT and NEG

- The NOT instruction inverts all bits of a byte, word, or doubleword. One's complement.
  - None of flags affected.
- NEG two's complements a number.
  - the arithmetic sign of a signed number changes from positive to negative or negative to positive
  - The CF flag cleared to 0 if the source operand is 0; otherwise it is set to 1. other flags are set according to the result.
- The NOT function is considered logical, NEG function is considered an arithmetic operation.
- NOT and NEG can use any addressing mode except segment register addressing.

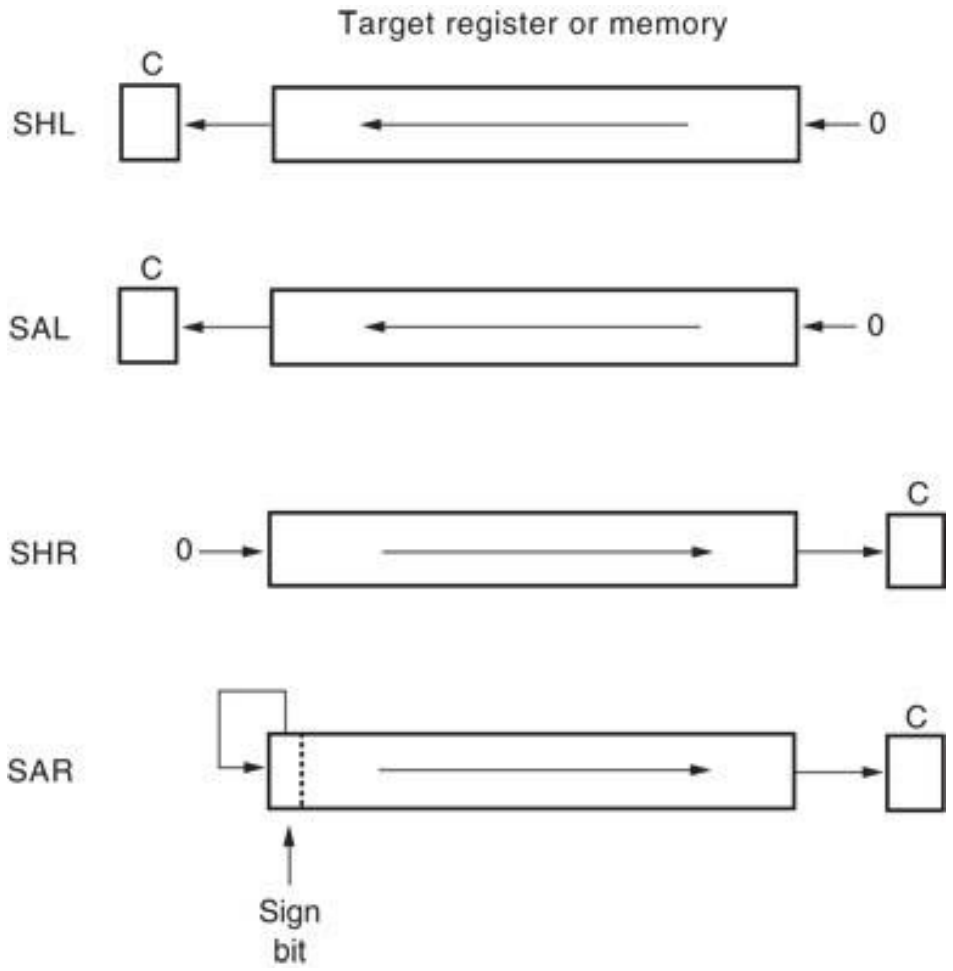
Assembly Language	*Operation*
NOT CH	CH is one's complemented
NEG CH	CH is two's complemented
NEG AX	AX is two's complemented
NOT EBX	EBX is one's complemented
NEG ECX	ECX is two's complemented
NOT RAX	RAX is one's complemented (64-bit mode)
NOT TEMP	The contents of data segment memory location TEMP is one's complemented
NOT BYTE PTR[BX]	The byte contents of the data segment memory location addressed by BX are one's complemented

Shift and Rotate

- Shift and rotate instructions manipulate binary numbers at the binary bit level.
  - as did AND, OR, Exclusive-OR, and NOT
- Common applications in low-level software used to control I/O devices.
- The microprocessor contains a complete complement of shift and rotate instructions that are used to shift or rotate any memory data or register.

Shift

- Position or move numbers to the left or right within a register or memory location.
  - also perform simple arithmetic as multiplication by powers of 2^{+n} (left shift) and division by powers of 2^{-n} (right shift).
- The microprocessor's instruction set contains four different shift instructions:
  - two are logical; two are arithmetic shifts
- All four shift operations appear in Figure .



- logical shifts move 0 in the rightmost bit for a logical left shift;
- The arithmetic shift left is identical to the logical shift left.
- 0 to the leftmost bit position for a logical right shift
- arithmetic right shift copies the sign-bit through the number.

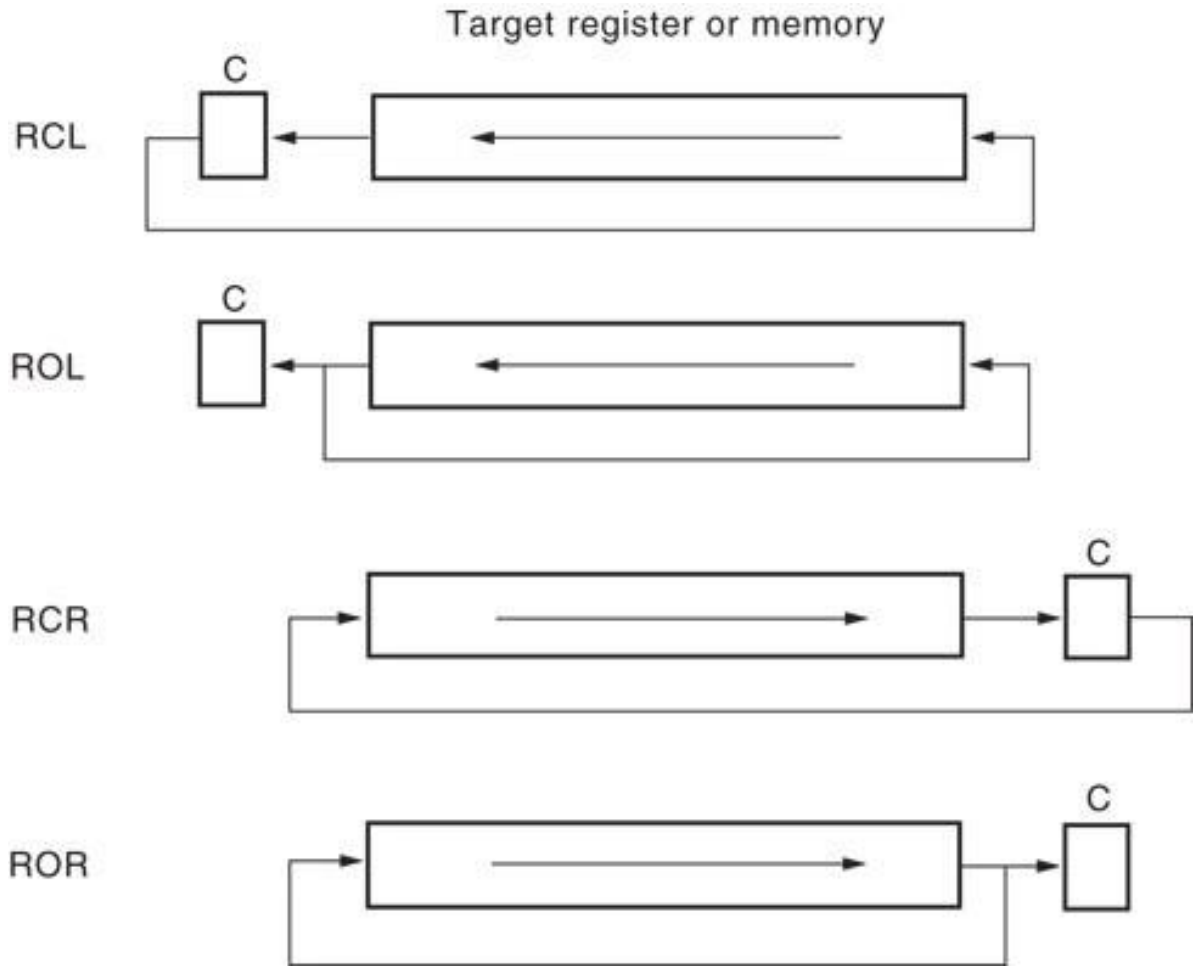
- Logical shifts multiply or divide unsigned data, arithmetic shifts multiply or divide signed data.
  - a shift left always multiplies by 2 for each bit position shifted
  - a shift right always divides by 2 for each position

Assembly Language	*Operation*
SHL AX,1	AX is logically shifted left 1 place
SHR BX,12	BX is logically shifted right 12 places
SHR ECX,10	ECX is logically shifted right 10 places
SHL RAX,50	RAX is logically shifted left 50 places (64-bit mode)
SAL DATA1,CL	The contents of data segment memory location DATA1 are arithmetically shifted left the number of spaces specified by CL
SHR RAX,CL	RAX is logically shifted right the number of spaces specified by CL (64-bit mode)
SAR SI,2	SI is arithmetically shifted right 2 places
SAR EDX,14	EDX is arithmetically shifted right 14 places

Rotate

- Positions binary data by rotating information in a register or memory location, either from one end to another or through the carry flag.
  - used to shift/position numbers wider than 16 bits
- With either type of instruction, the programmer can select either a left or a right rotate.
- Addressing modes used with rotate are the same as those used with shifts

- A rotate count can be immediate or located in register CL.
  - if CL is used for a rotate count, it does not change
- Rotate instructions are often used to shift wide numbers to the left or right.



Assembly Language	*Operation*
ROL SI,14	SI rotates left 14 places
RCL BL,6	BL rotates left through carry 6 places
ROL ECX,18	ECX rotates left 18 places
ROL RDX,40	RDX rotates left 40 places
RCR AH,CL	AH rotates right through carry the number of places specified by CL
ROR WORD PTR[BP],2	The word contents of the stack segment memory location addressed by BP rotate right 2 places

Bit Scan Instructions

- Scan through a number searching for the first 1-bit.
  - accomplished by shifting the number
  - available in 80386–Pentium 4
- BSF scans the number from the leftmost bit toward the right; BSR scans the number from the rightmost bit toward the left.
  - if a 1-bit is encountered, the zero flag is set and the bit position number of the 1-bit is placed into the destination operand
  - if no 1-bit is encountered the zero flag is cleared

String Comparisons

- String instructions are powerful because they allow the programmer to manipulate large blocks of data with relative ease.
- Block data manipulation occurs with MOVS, LODS, STOS, INS, and OUTS.
- Additional string instructions allow a section of memory to be tested against a constant or against another section of memory.

  SCAS (string scan); CMPS (string compare)

Program Control Instructions

Hardware Lab

Introduction

- Program Control Instructions direct the flow of a program and allow the flow to change

- Change in flow occurs after a decision, made with CMP or TEST instruction, followed by a conditional jump instruction

THE JUMP GROUP

- Allows programmer to skip program sections and branch to any part of memory for next instruction.

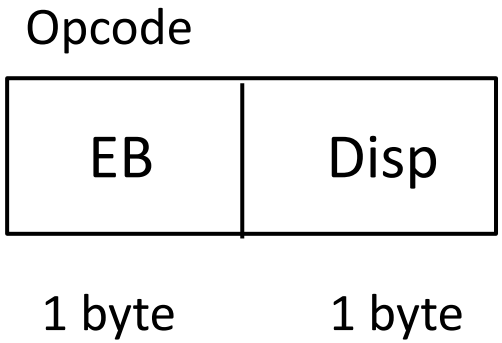
- A conditional jump instruction allows decisions based upon numerical tests.
  - results are held in the flag bits, then tested by conditional jump instructions
- An unconditional Jump instruction does not depend on any condition or numerical tests.

Unconditional Jump (JMP)

- Three types: short jump, near jump, far jump.
- The short and near jumps are often called **intrasegment jumps.**
- Far jumps are called **intersegment jumps.**

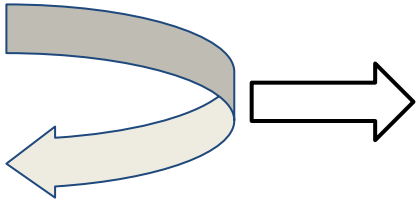
Short Jump

- Called **relative jumps**
- Jump address is not stored with the opcode
- A distance, or displacement, follows the opcode
- The short jump displacement is a distance represented by a 1-byte signed number
- It allows jumps or branches to memory location within +127 and -128 bytes from the address following the jump.




```
0000      XOR BX,BX
0002      START:   MOV AX,1
0005      ADD AX,BX
0007      JMP SHORT NEXT

0020      NEXT:    MOV BX,AX
0022      JMP START
```



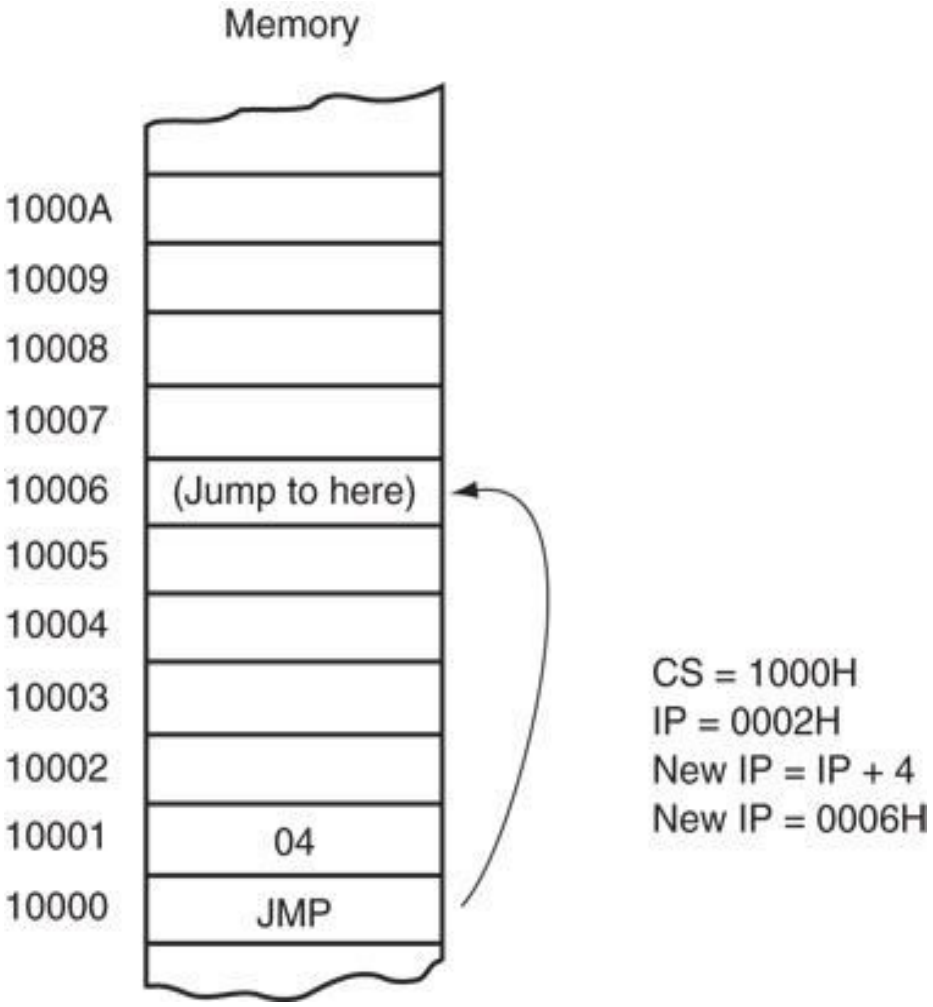
Disp= 0020H – 0009H

= 0017 (17H)

A short jump to four memory locations beyond the address of the next instruction.

Example:

JMP 04H



Near Jump

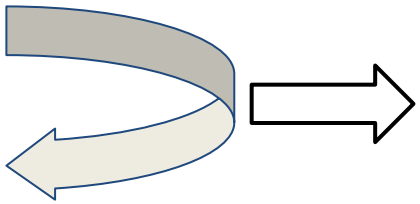
- A near jump passes control to an instruction in the current code segment located within ±32K bytes from the near jump instruction.
- 3-byte instruction with opcode followed by a signed 16-bit displacement.
- Signed displacement is added to the instruction pointer (IP) to generate the jump address.
- can jump to any memory location within the current real mode code segment

Opcode

E9	Disp low	Disp high
1 byte	1 byte	1 byte

```
0000      XOR BX,BX
0002      START:    MOV AX,1
0005      ADD AX,BX
0007      JMP SHORT NEXT

0200      NEXT:     MOV BX,AX
0202      JMP START
```

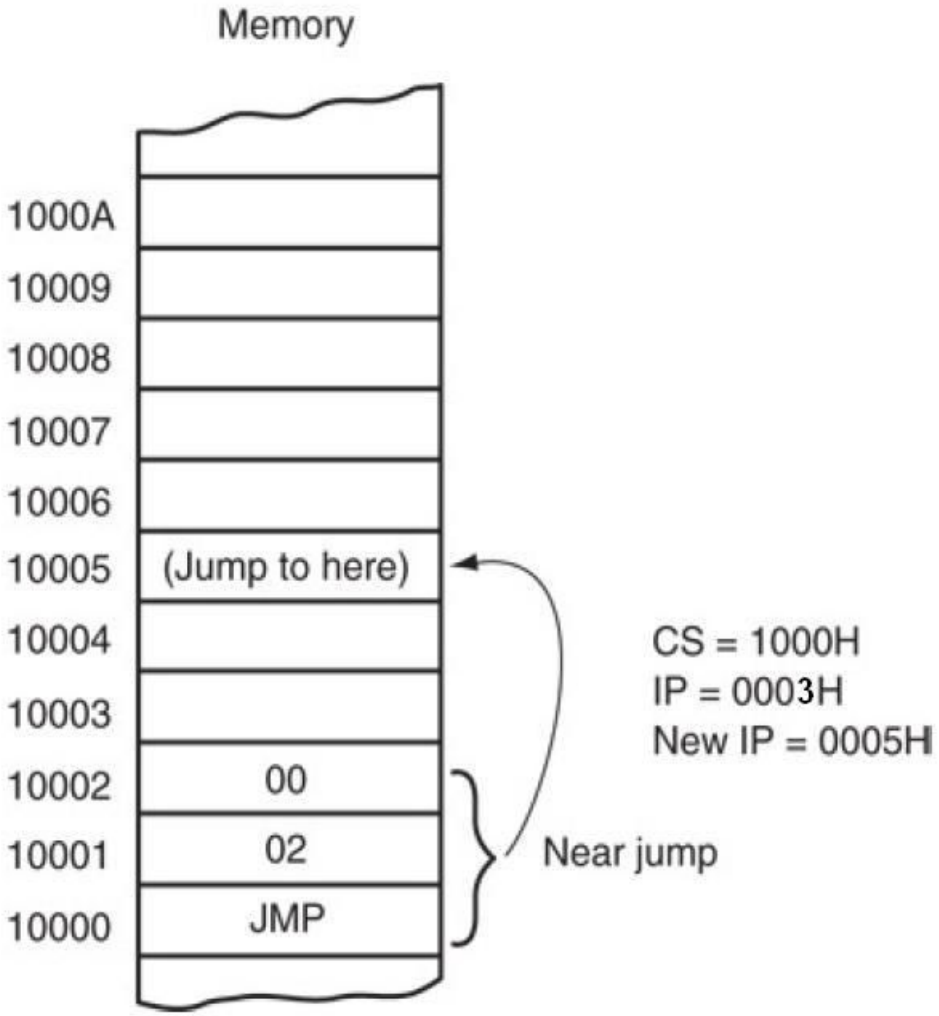


Disp= 0200H – 000AH

= 01F6H

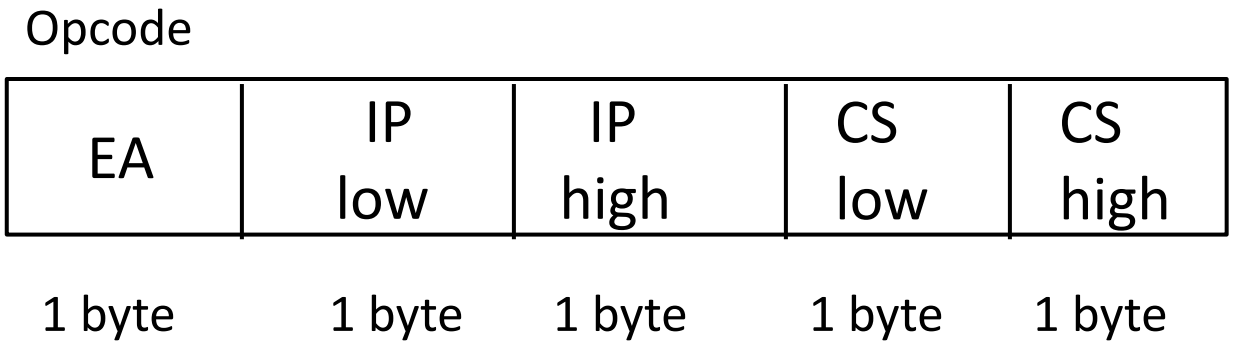
A near jump that adds the displacement (0002H) to the contents of IP.

Example:
JMP 0002H



Far Jump

- 5-byte instruction
- Bytes 2 and 3 of this instruction contain the new offset address
- Bytes 4 and 5 contain the new segment address
- offset address (16 or 32 bits) - contains the offset address within the new code segment



Far Jump

- Far jump instruction appears with the FAR PTR directive
- OR define a label as a *far label.*
- EXTRN UP:FAR -> defines the label UP as a far label
- A label is far only if it is external to the current code segment or procedure.
- The JMP UP instruction in the example references a far label.

Example:

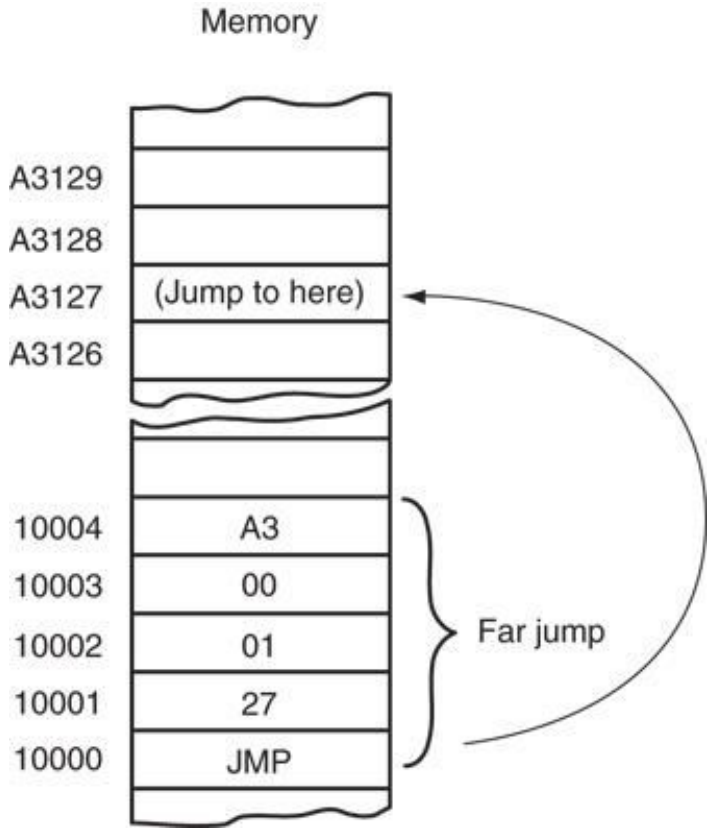
```
EXTRN UP:FAR
      XOR BX,BX
START: ADD AX,1
 JMP NEXT

NEXT:  MOV BX,AX
 JMP FAR PTR START
JMP UP
```

A far jump instruction replaces the contents of both CS and IP with 4 bytes following the opcode.

IP= 10005
New offset= 0127
New CS=A300
New IP= A3000+0127=A3127

Example:
JMP 0127: A300



Jumps with Register Operands

- Jump can use a 16- or 32-bit register as an operand.
- automatically sets up as an **indirect jump**
- allows a jump to any location within the current code segment
- Register content (address of the jump) is transferred directly into IP
- JMP AX - copies the contents of AX register into the IP

Conditional Jumps

- Always short jumps in 8086 - 80286.
- In 80386 and above, conditional jumps are short or near jumps
- Jump to any location within the current code segment.
- Conditional jump instructions test flag bits:
  - sign (S), zero (Z), carry (C), parity (P), overflow (O)

- Condition = true branch to label
- Condition = false execute the next sequential step in program

TABLE 6–1 Conditional jump instructions.

Assembly Language	*Tested Condition*	*Operation*
JA	Z = 0 and C = 0	Jump if above
JAE	C = 0	Jump if above or equal
JB	C = 1	Jump if below
JBE	Z = 1 or C = 1	Jump if below or equal
JC	C = 1	Jump if carry
JE or JZ	Z = 1	Jump if equal or jump if zero
JG	Z = 0 and S = 0	Jump if greater than
JGE	S = 0	Jump if greater than or equal
JL	S != O	Jump if less than
JLE	Z = 1 or S != O	Jump if less than or equal
JNC	C = 0	Jump if no carry
JNE or JNZ	Z = 0	Jump if not equal or jump if not zero
JNO	O = 0	Jump if no overflow
JNS	S = 0	Jump if no sign (positive)
JNP or JPO	P = 0	Jump if no parity or jump if parity odd
JO	O = 1	Jump if overflow
JP or JPE	P = 1	Jump if parity or jump if parity even
JS	S = 1	Jump if sign (negative)
JCXZ	CX = 0	Jump if CX is zero
JECXZ	ECX = 0	Jump if ECX equals zero
JRCXZ	RCX = 0	Jump if RCX equals zero (64-bit mode)

Conditional Jumps

- To compare signed numbers, use
  - JG, JL, JGE, JLE, JE, and JNE instructions.
- To compare unsigned numbers, use
  - JA, JB, JAB, JBE, JE, and JNE instructions.
- Remaining conditional jumps test individual flag bits, such as overflow and parity.

Conditional Set

- 80386 onwards
- set a byte to either 01H or clear a byte to 00H, depending on the outcome of the condition under test
- useful where a condition must be tested at a later point in the program.
- SETNC MEM instruction - places 01H into memory location MEM if carry is cleared, and 00H into MEM if carry is set.

TABLE 6–2 Conditional set instructions.

Assembly Language	*Tested Condition*	*Operation*
SETA	Z = 0 and C = 0	Set if above
SETAE	C = 0	Set if above or equal
SETB	C = 1	Set if below
SETBE	Z = 1 or C = 1	Set if below or equal
SETC	C = 1	Set if carry
SETE or SETZ	Z = 1	Set if equal or set if zero
SETG	Z = 0 and S = 0	Set if greater than
SETGE	S = 0	Set if greater than or equal
SETL	S != O	Set if less than
SETLE	Z = 1 or S != O	Set if less than or equal
SETNC	C = 0	Set if no carry
SETNE or SETNZ	Z = 0	Set if not equal or set if not zero
SETNO	O = 0	Set if no overflow
SETNS	S = 0	Set if no sign (positive)
SETNP or SETPO	P = 0	Set if no parity or set if parity odd
SETO	O = 1	Set if overflow
SETP or SETPE	P = 1	Set if parity or set if parity even
SETS	S = 1	Set if sign (negative)

LOOP

- A combination of a decrement CX and the JNZ conditional jump.
- In 8086 - 80286 LOOP decrements CX.
  - if CX != 0, jumps to the address indicated by the label
  - If CX = 0, the next sequential instruction executes

LOOP

Example:

```
MOV CX, 25H

MOV AX, 1H

MOV BX, 4H

XXX : ADC AX, BX

      ADD BX, 3H

      DEC CX

      JNZ XXX // jump if result (value of CX) not zero
```

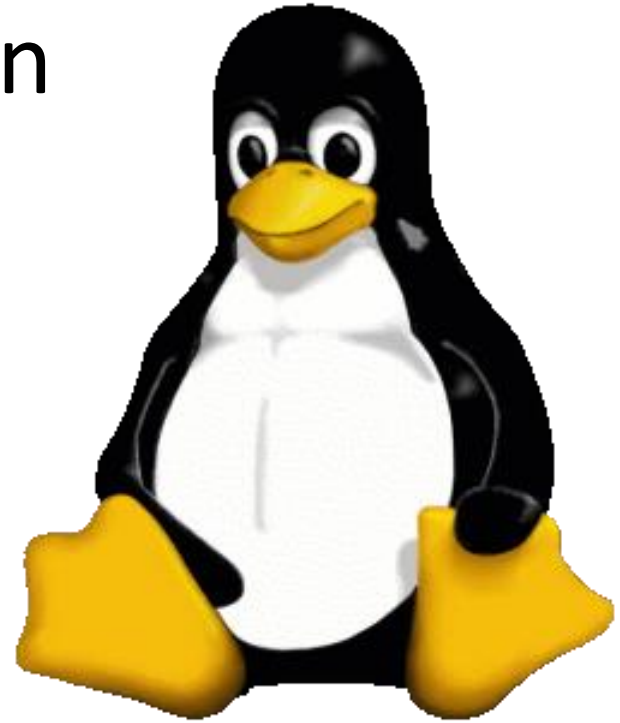
Conditional LOOPs

- LOOPE (loop while equal) instruction
  - jumps if CX != 0 while an equal condition exists.
  - will exit loop if the condition is not equal or CX register decrements to 0

- LOOPNE (loop while not equal)
  - jumps if CX != 0 while a not-equal condition exists.
  - will exit loop if the condition is equal or the CX register decrements to 0

Thank You

Interrupts & System Calls implementation in Linux



Outline

- Interrupt
- Linux OS & System Calls
- NASM Programming
  - Program structure
  - Declaring variables
  - I/O operations
  - Sample Programs

Question??

- When you are at home sitting on your lazy boy, how do you know when someone wants to talk to you on the phone?

  - Do you periodically get up and pick up the phone to see if someone is there?
  - Or do you wait till the phone rings to answer it?

Answer?

- The first scenario shows a person doing what is known as polling.
- The second case illustrates an interrupt-driven person.

Interrupt

- Device sends a special signal to CPU when data arrives.
- “Wake me up when we get there.”
- Responds to hardware interrupt signal by interrupting current processing.
- Now CPU can perform tasks before and after interrupt instead of just polling!! Good!

- Interrupt is an asynchronous event which can take the execution control of the CPU

- Steps in interrupt execution

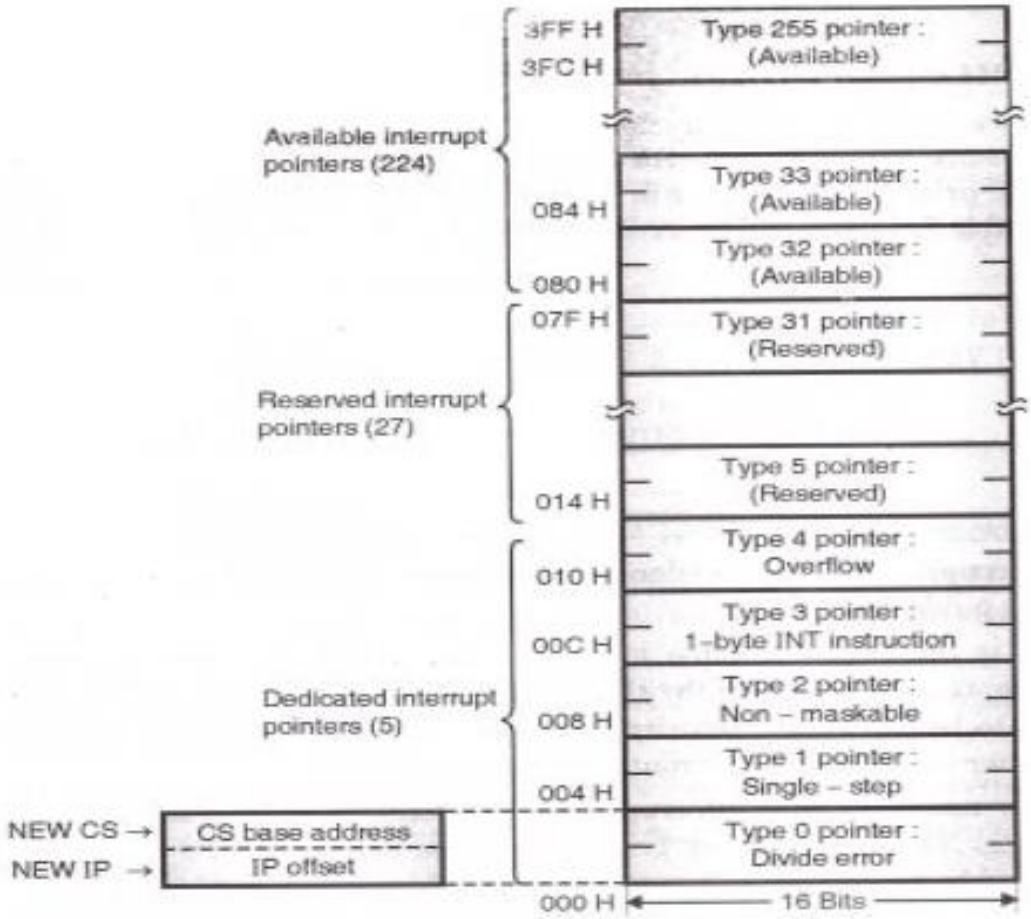
1. Completes current instruction
2. Saves current state to status registers
3. Identify the source
4. Jump to and activate Interrupt Service Routing (ISR)
5. Return to original program and restore state

IVT

- An "interrupt vector table" (IVT) is a **data structure that associates** a list of interrupt handlers with a list of interrupt requests in a table of interrupt vectors.
- An entry in the interrupt vector is the address of the interrupt handler.
- The first 1Kbyte of memory of 8086 (00000 to003FF) is set aside as a table for storing the starting addresses of Interrupt Service Procedures (ISP).
- The starting address of an **ISP** is often called the **Interrupt Vector** or Interrupt Pointer. Therefore the table is referred as **Interrupt Vector Table**

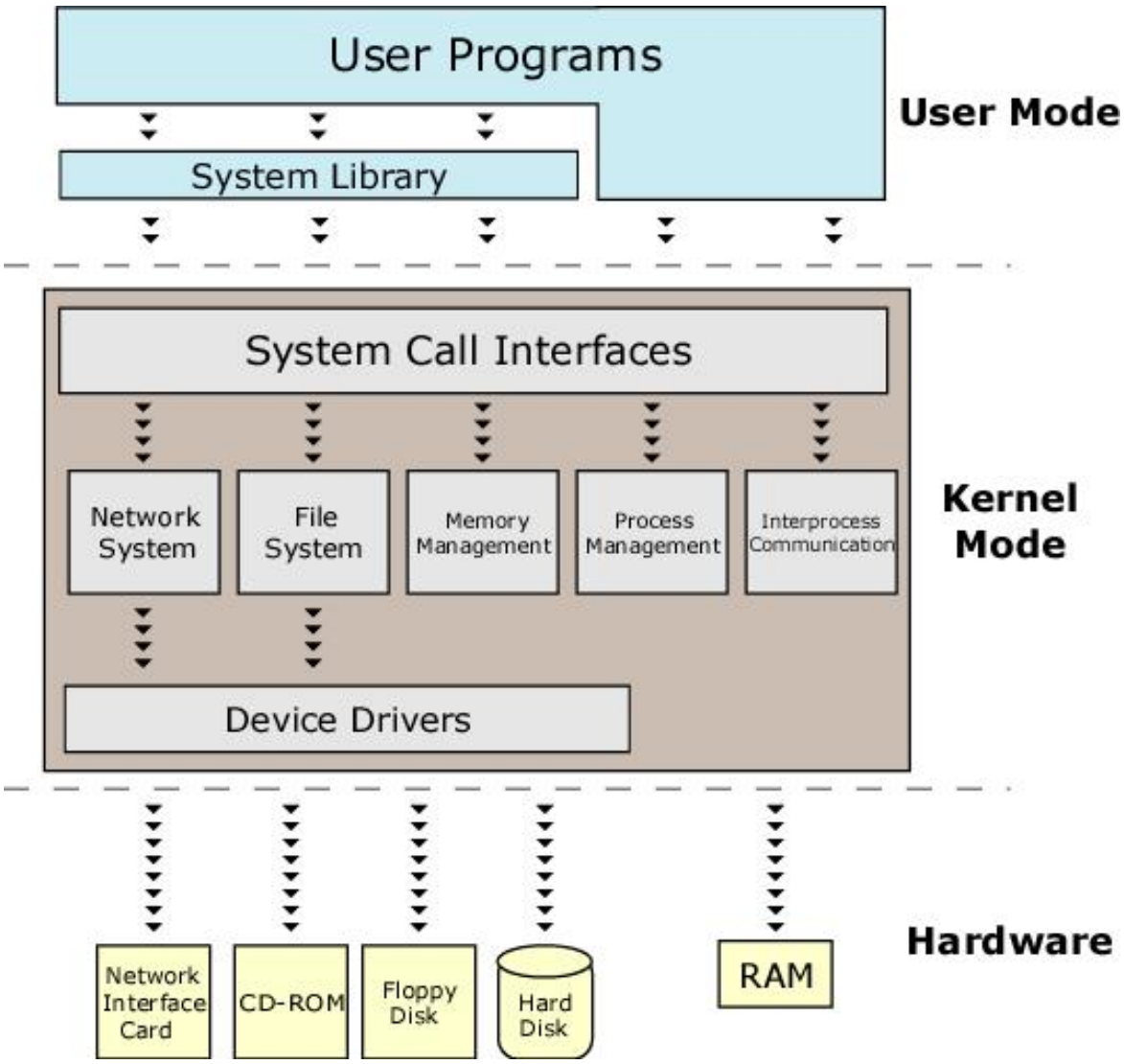
Vector Range

Vector range	Use
0–19 (0x0–0x13)	Nonmaskable interrupts and exceptions
20–31 (0x14–0x1f)	Intel-reserved
32–127 (0x20–0x7f)	External interrupts (IRQs)
128 (0x80)	Programmed exception for system calls
129–238 (0x81–0xee)	External interrupts (IRQs)

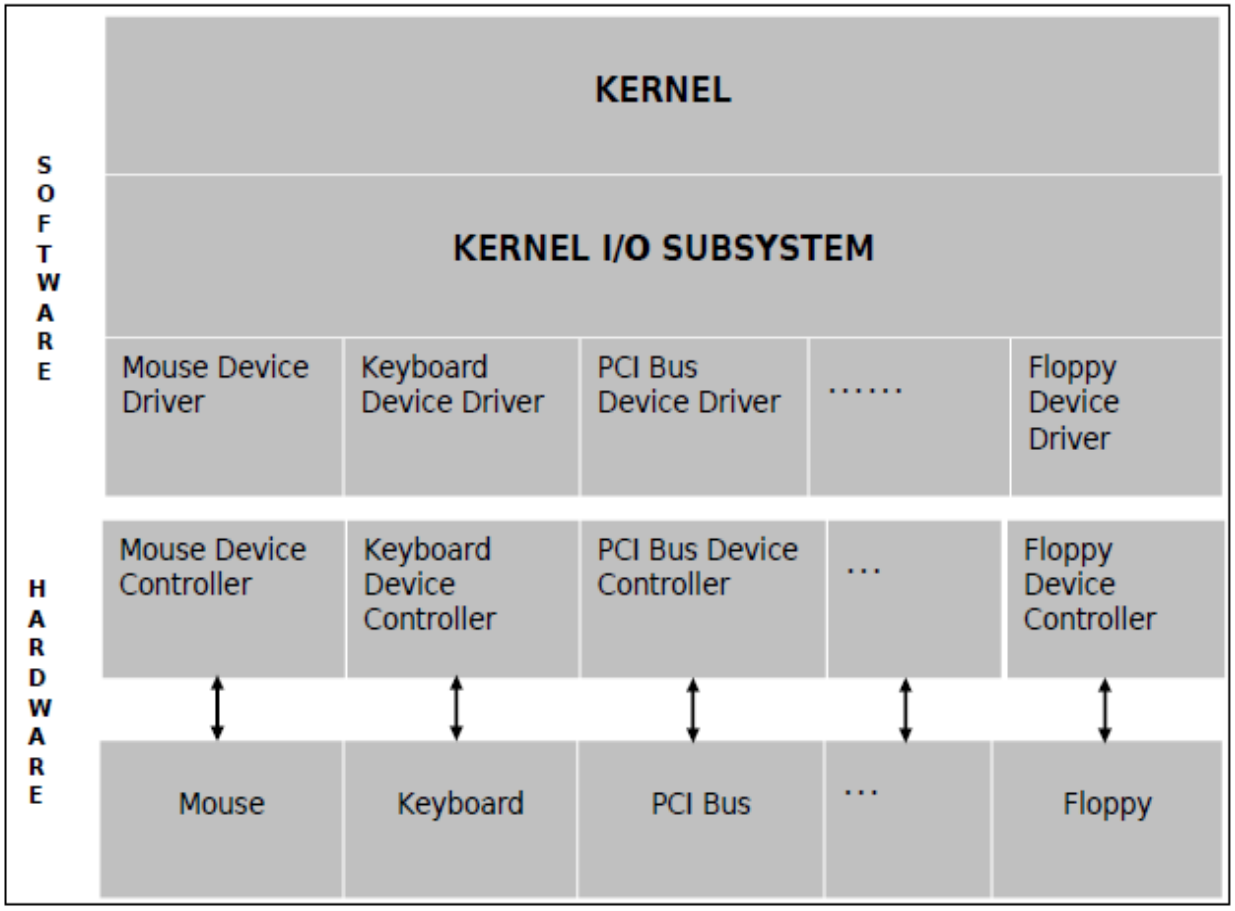


Interrupt vector table

Structure of Linux OS

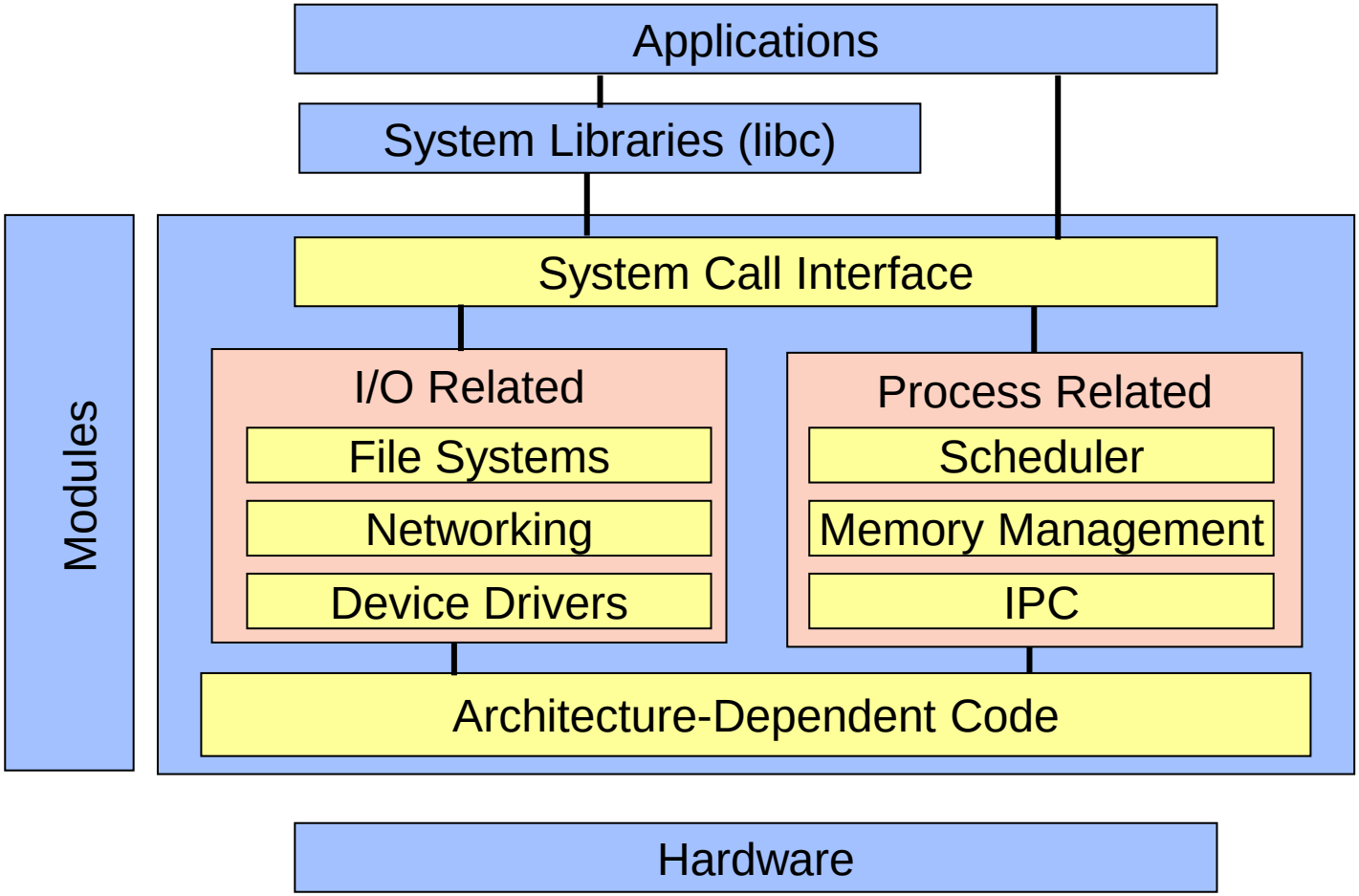

User Programs
System Library
User Mode
System Call Interfaces
Network System
File System
Memory Management
Process Management
Interprocess Communication
Device Drivers
Kernel Mode
Network Interface Card
CD-ROM
Floppy Disk
Hard Disk
RAM
Hardware

Split View



Split View of Kernel I/O SubSystem with Device Drivers

Linux Architecture



System Calls

- Interface between user-level processes and hardware devices like CPU, memory, disks etc.
- Make programming easier:
  - Let kernel take care of hardware-specific issues.
- Increase system security:
  - Let kernel check requested service via system call.
- Provide portability:
  - Maintain interface but change functional implementation.

System calls

- System calls (or syscalls) are function calls made from user space programs into the kernel requesting for some service

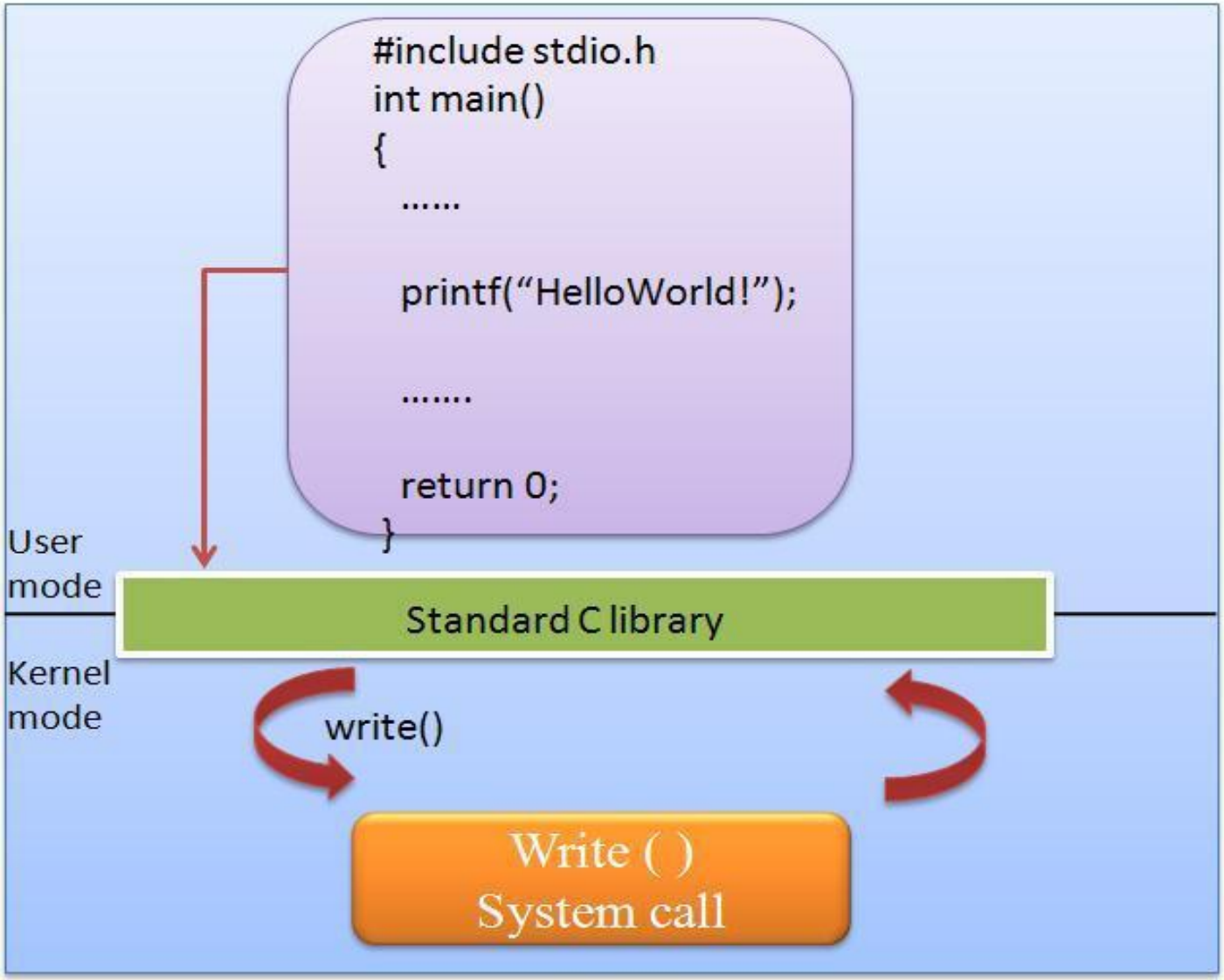
Examples:

- Open( ),read( ), write( ), fork( ), lseek( ), clone( ), wait ( ), etc..

Application using Standard Library Interface

```
#include <stdio.h>
int main()
{
    printf(“Hello World\n”);
    return(0);
}
```

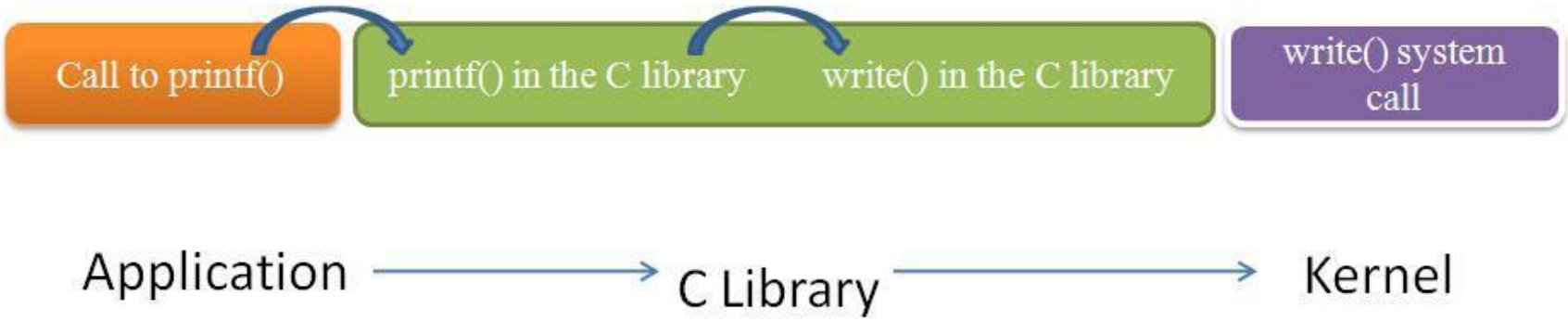
Standard library Vs system call



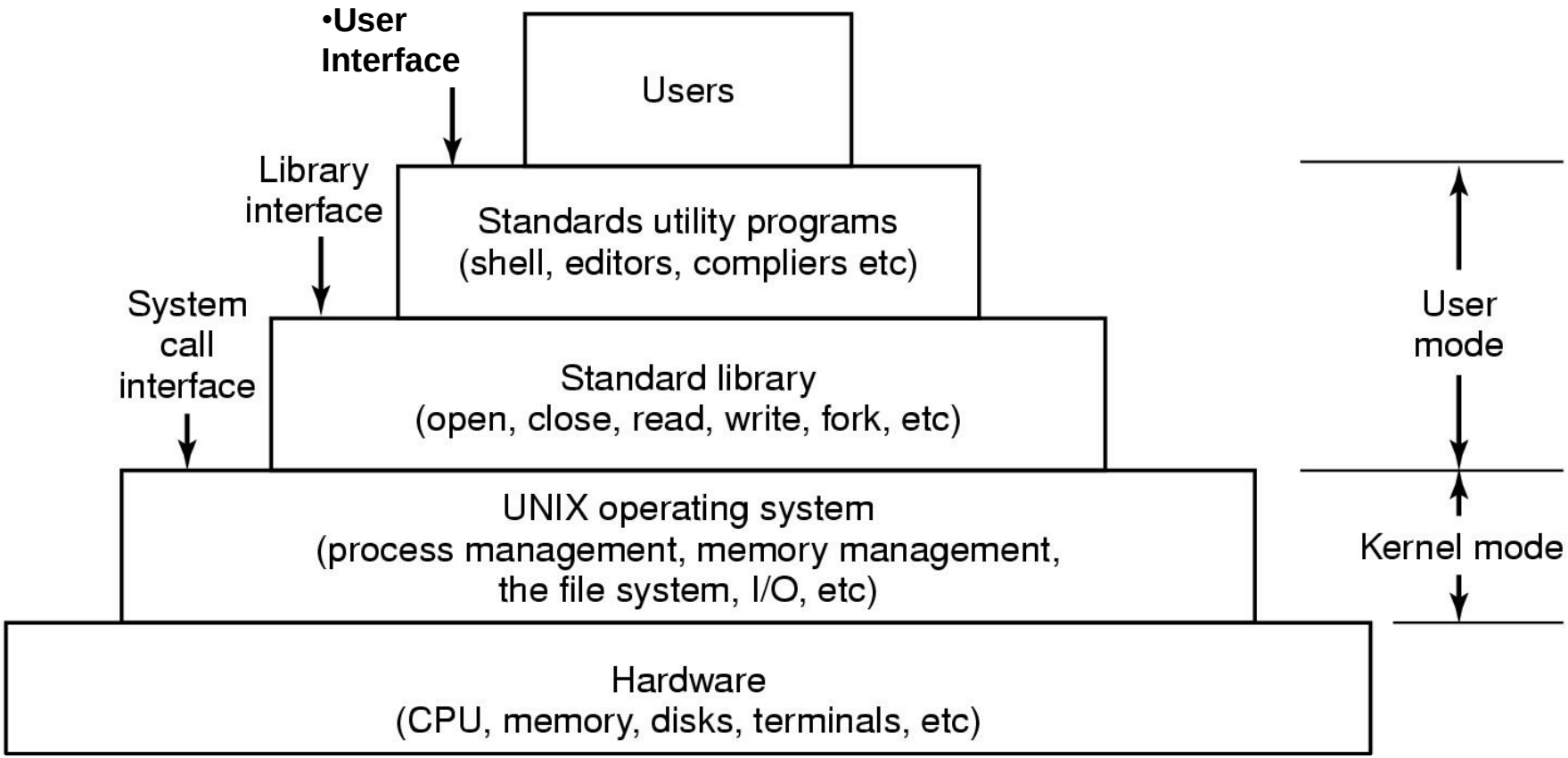
Application using System Call Interface

```
include <unistd.h>
int main()
{
    write(1,”Hello World\n”, 12);
    return(0);
}
```

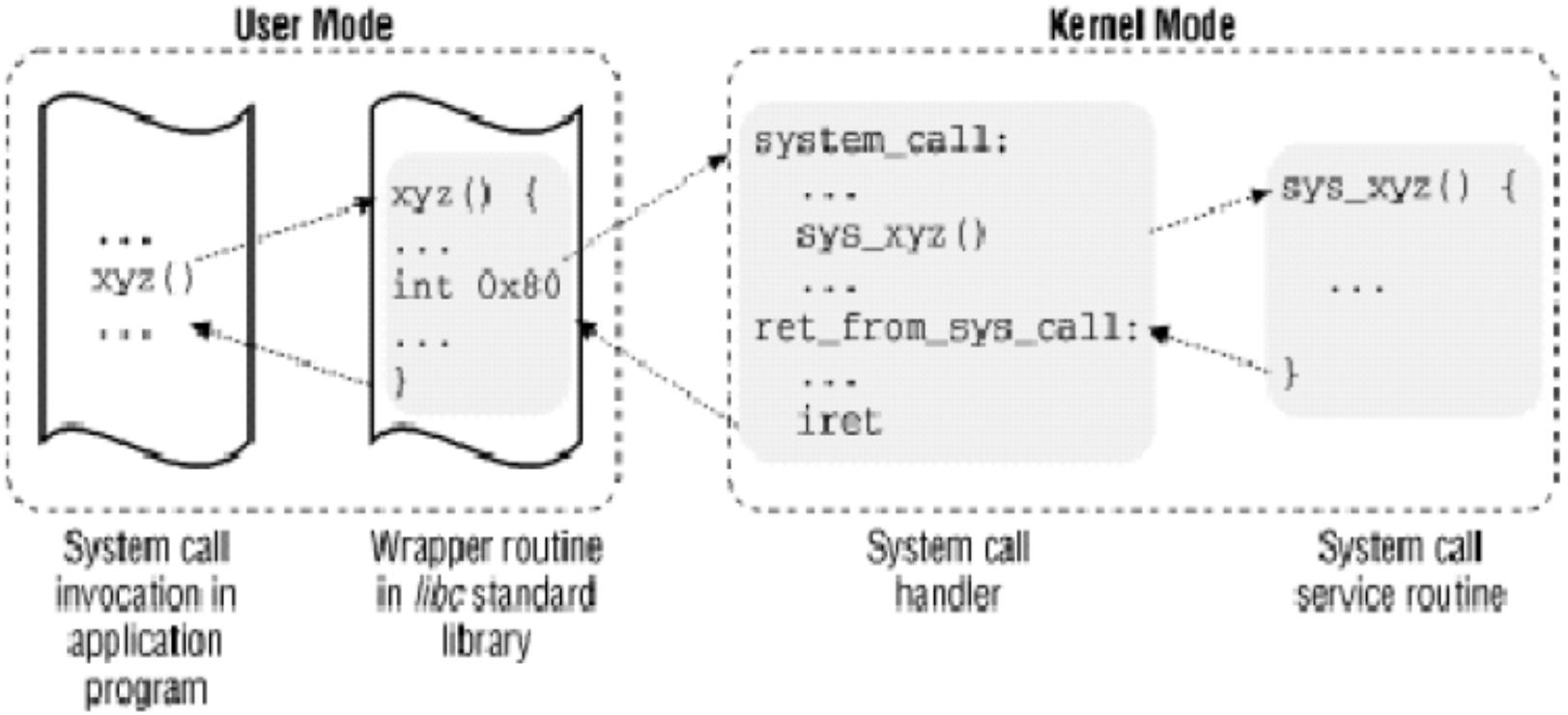
Application ->C Library ->KernelCentre



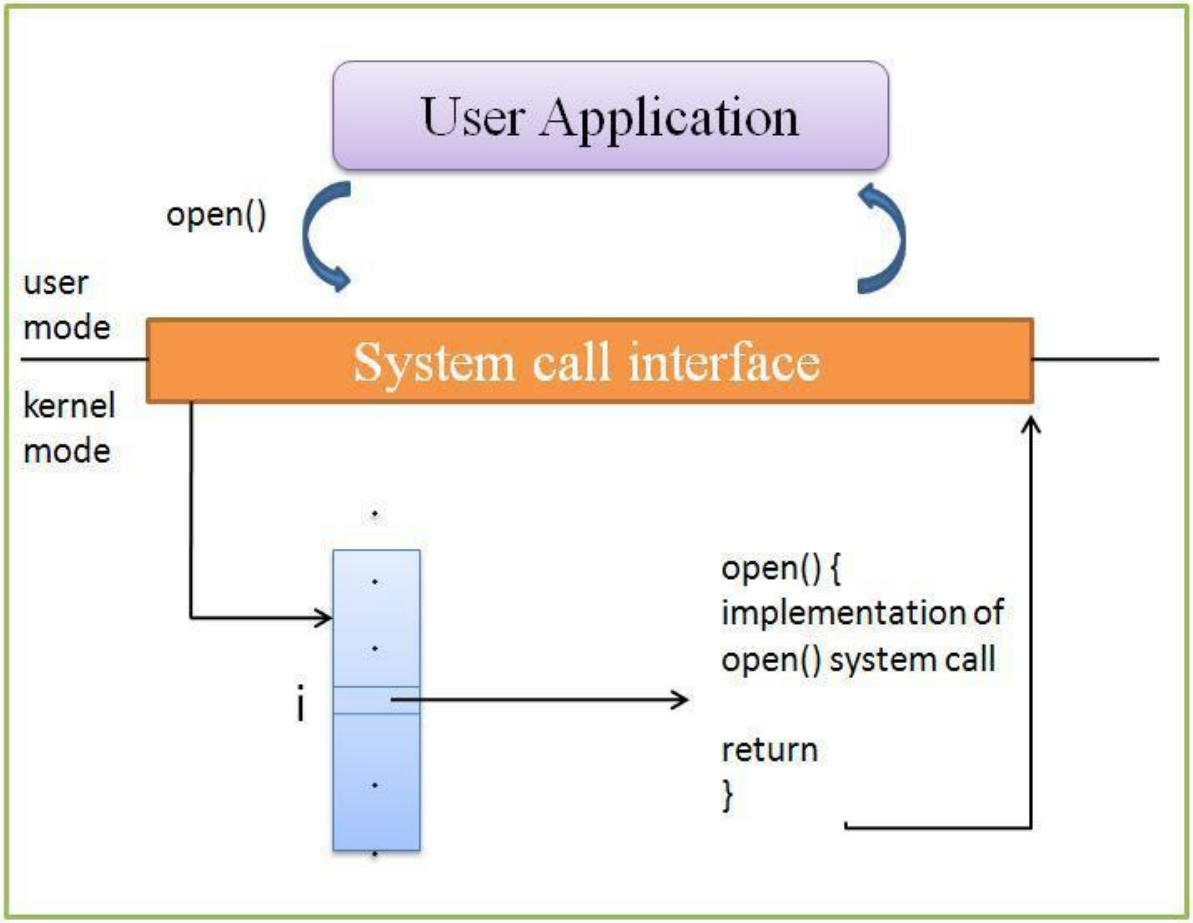
The layers of a LINUX system



Invoking a System Call



Access to kernel services



System Call Handling

- User-space applications cannot execute kernel code directly
- The mechanism to signal the kernel is a **software interrupt**
- Occurs through an exception and then the system will switch to kernel mode and execute the exception handler

System call Handling

- Each system call is assigned a **syscall number**.
- The kernel keeps a list of all registered system calls in the system call table, stored in **sys_call_table.**
- On x86, the **syscall number** is fed to the kernel via the **eax** register.

Linux System Calls

Invoked by executing **`int $0x80`**.

- Programmed exception vector number 128.
- CPU switches to kernel mode & executes a kernel function.

- Calling process passes **`syscall number`** identifying system call in **`eax`** register (on Intel processors).
- Syscall handler responsible for:
  - Saving registers on kernel mode stack.
  - Invoking syscall service routine.

Parameter Passing

- On the 32-bit Intel 80x86:
  - 6 registers are used to store syscall parameters.
    - **`eax`** (syscall number).
    - **`ebx`**, **`ecx`**, **`edx`**, **`esi`**, **`edi`** store parameters to syscall service routine, identified by syscall number.

Parameter passing

- In case of six or more arguments, a single register is used to hold a pointer to user-space where all the parameters are stored
- The return value is sent to user-space also via register. On x86, it is written into the eax register

NASM Programming

Chapter 3,4 & 5 NASM manual

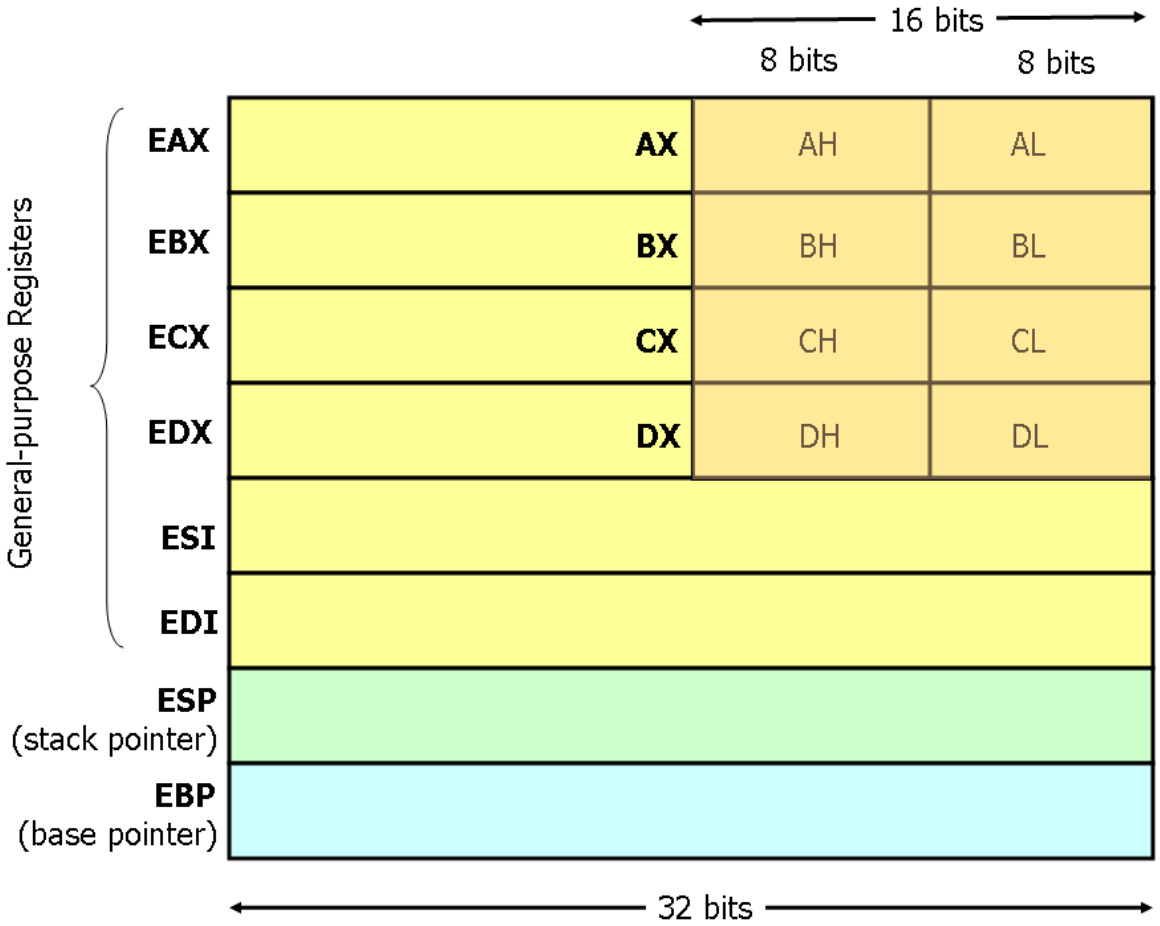
Why Assembly Language ?

- you will get a better idea of computer organization and how a program executes in a computer.
- A program written in assembly language will be more efficient than the same program written in a high level language.
- Some portions of Operating System (Eg: Linux kernel) and some system software are written in assembly language.

X86 architecture

- The x86 architecture is an Instruction Set Architecture (ISA) series for computer processors.
- Developed by Intel Corporation, x86 architecture defines how a processor handles and executes different instructions passed from the operating system (OS) and software programs.
- The ’x’ in x86 denotes ISA version.

X86(i386) Registers



NASM

- The Netwide Assembler is an assembler and disassembler for the Intel X86 architecture
- NASM is considered to be one of the most popular assemblers for Linux.

NASM Installation

- NASM is freely available on internet.

In Ubuntu Linux you can give the command :

  – sudo apt-get install nasm

and in fedora you can use the command:

  – su -c yum install nasm

in a terminal and easily install nasm

Sections in NASM

1. section .text :

This section contains the executable code from where execution starts. It is analogous to the main() function in C.

2. section .bss :

Here, variables are declared without initialization.

3. section .data :

Variables are declared and initialized in this section

declaring space in the memory

(a) RESx:

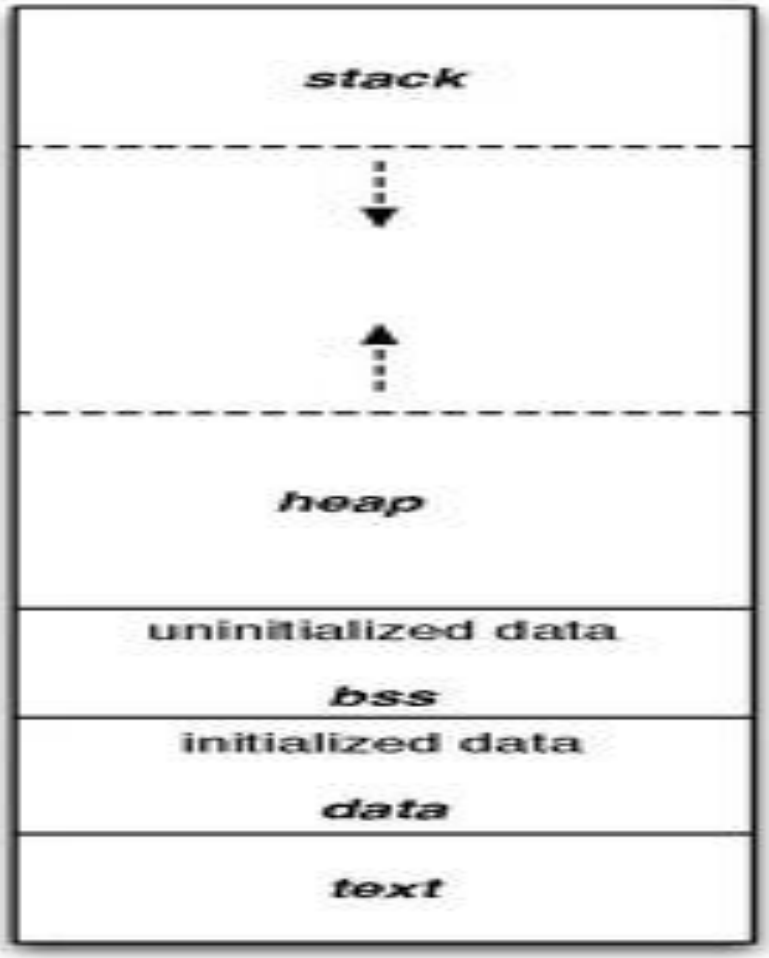
Reserve just space in memory for a variable without giving any initial values.

(b) Dx:

Declaring space in the memory for any variable and also providing the initial values at that moment. Where x can be replaced with different characters as shown in the below table.

x	Meaning	Bytes
b	BYTE	1
w	WORD	2
d	DOUBLE WORD	4
q	QUAD WORD	8
t	TEN BYTE	20

RAM memory layout



```
section .data
var1: db 10  ;Reserve one byte in memory for storing var1 and var1=10
var2 : db 1,2,3,4
string: db 'Hello'
string2: db 'H','e','l','l','o'
```

bss stands for block starting symbol

```
section .bss
var1: resb 1
var2: resq 1
var3: resw 1
```

Compilation

- **Assembling the source file**
  - **nasm -f elf filename.asm**
  - This creates an object file, filename.o in the current working directory.
- **Creating an executable file**
  - ld filename.o -o output_filename (32 bit machine)
  - ld -melf_i386 filename.o -o output_filename (64bit)
- This creates an executable file of the name output_filename.
- **Program execution**
  - ./output_filename

Sample execution

- For example, if the program to be run is first.asm
  - nasm -f elf first.asm
  - ld first.o -o output
  - ./output

Basic I/O in NASM

- NASM does not support I/O operations directly
- I/O in NASM Program is implemented using the Operating System's read and write system call.
- Interrupt no: 80h is as the software generated interrupt in Linux Systems.
- Applications implement the System Calls using this interrupt.
- **When an application triggers int 80h, then OS will understand that it is a request for a system call and it will refer the general purpose registers to find out and execute the exact Interrupt Service Routine**

1. Exit System Call

```
mov eax, 1 ; System Call Number
mov ebx, 0 ; Parameter
int 80h ; Triggering OS Interrupt
```

Read System Call

```
mov eax, 3 ; Sys_call number for read
mov ebx, 0 ; Source Keyboard
mov ecx, var ; Pointer to memory location
mov edx, dword[size] ; Size of the string
int 80h ; Triggering OS Interrupt
```

- This method is also used for reading integers and it is bit tricky. If we need to read a single digit, we will read it as a single character and then subtract 30h from it(ASCII of 0 = 30h). Then we will get the actual value of that number in that variable.

Reading single Digit

```
mov eax, 3
mov ebx, 0
mov ecx, digit1
mov edx, 1
int 80h
sub byte[digit1], 30h ;Now we have the actual number in [digit1]
```

Reading Multi digit number

```
while(temp!=enterkey)
   num=0
   read(temp)
   num=numx10+temp
endwhile
```

Reading a two digit number

```
mov eax, 3
mov ebx, 0
mov ecx, digit1
mov edx, 1
int 80h

;Reading second digit
mov eax, 3
mov ebx, 0
mov ecx, digit2
mov edx, 2              ;Here we put 2 because we need to read and
int 80h                  omit enter key press as well

sub byte[digit1], 30h
sub byte[digit2], 30h

;Getting the number from ASCII
; num = (10* digit1) + digit2

mov al, byte[digit1]     ; Copying first digit to al
mov bl, 10
mul bl                   ; Multiplying al with 10
movzx bx, byte[digit2]   ; Copying digit2 to bx
add ax, bx

mov byte[num], al        ; We are sure that no less than 256, so we can
omit higher 8 bits of the result.
```

Write System Call

```
mov eax, 4 ;Sys_call number
mov ebx, 1 ;Standard Output device
mov ecx, msg1 ;Pointer to output string
mov edx, size1 ;Number of characters
int 80h ;Triggering interrupt.
```

- This method is even used to output numbers. If we have a number we will break that into digits. Then we keep each of that digit in a variable of size 1 byte. Then we add 30h (ASCII of 0) to each, doing so we will get the ASCII of character to be print.

Hello world program

```
_start:
mov eax, 4                ; Using int 80h to implement write() sys_call
mov ebx, 1
mov ecx, string
mov edx, length
int 80h

;Exit System Call
mov eax, 1
mov ebx, 0
int 80h

section .data                        ;For Storing Initialized Variables
string: db 'Hello World', 0Ah        ;String Hello World followed by a
newline character
length: equ 13                       ; Length of the string stored to a constan
```

nasm -f elf first.asm

ld first.o -o output

./output

- The **text** section is used for keeping the actual code. This section must begin with the declaration **global _start**, which tells the kernel where the program execution begins.
- The syntax for declaring text section is –

```
Section .text
  global _start
_start:
```

Printing a multi digit number

- **Pseudo Code**

```
count=0
while (num!=0)
    temp=num%10
    count++
    push(temp)
    num=num/10
endwhile
while(count!=0)
    temp=pop()
    print(temp)
    count- -
```

```
print_num:
mov byte[count],0
pusha

extract_no:
cmp word[num], 0
je print_no
inc byte[count]
mov dx, 0
mov ax, word[num]

mov bx, 10
div bx
push dx
mov word[num], ax
jmp extract_no

print_no:
cmp byte[count], 0
je end_print
dec byte[count]
pop dx
mov byte[temp], dl
add byte[temp], 30h
mov eax, 4
mov ebx, 1
mov ecx, temp
mov edx, 1
int 80h
jmp print_no
```

```
end_print:
mov eax,4
mov ebx,1
mov ecx,newline
mov edx,1
int 80h
```