Instruction Set Architecture (Week#7)

(Interface between hardware and software)

Dr. Nausheen Shoaib

Intel x86 and ARM Operation Types

12.19 Convert the following formulas from infix to reverse Polish:

a. A + B + C + D + E

b. (A + B) × (C + D) + E

c. (A × B) + (C × D) + E

Intel x86 and ARM Operation Types

12.9 In Section 12.4, it was stated that both an arithmetic left shift and a logical left shift correspond to a multiplication by 2 when there is no overflow, and if overflow occurs, arithmetic and logical left shift operations produce different results, but the arithmetic left shift retains the sign of the number. Demonstrate that these statements are true for 5-bit twos complement integers.

Intel x86 and ARM Operation Types

d. Now consider the high-level language statement:

A: = (B > C) OR (D = F)

A compiler might generate the following code:

```
        MOV     EAX, B      ;move contents of location B to register EAX
        CMP     EAX, C      ;compare contents of register EAX and location C
        MOV     BL, 0       ;0 represents false
        JLE     N1          ;jump if (B ≤ C)
        MOV     BL, 1       ;1 represents false
N1      MOV     EAX, D
        CMP     EAX, F
        MOV     BH, 0
        JNE     N2
        MOV     BH, 1
N2      OR      BL, BH
```

Show an alternative implementation using the SETcc instruction that saves memory and execution time.

The comparison sets the processor flags based on the result of B - C:
•If B > C, the greater flag will be set.
•If B <= C, the less than or equal flag will be set.

sets the BL register to 0, representing false. We will use BL to store the result of the comparison (B > C)

This is a conditional jump instruction that checks the comparison flags set by the CMP instruction.
•If B <= C, the JLE (Jump if Less or Equal) condition is true, and the program jumps to label N1.
•If B > C, the jump is not taken, and the next instruction (MOV BL, 1) is executed

Intel x86 and ARM Operation Types

d. Now consider the high-level language statement:

A: = (B > C) OR (D = F)

A compiler might generate the following code:

```
        MOV     EAX, B      ;move contents of location B to register EAX
        CMP     EAX, C      ;compare contents of register EAX and location C
        MOV     BL, 0       ;0 represents false
        JLE     N1          ;jump if (B ≤ C)
        MOV     BL, 1       ;1 represents false
N1      MOV     EAX, D
        CMP     EAX, F
        MOV     BH, 0
        JNE     N2
        MOV     BH, 1
N2      OR      BL, BH
```

Show an alternative implementation using the SETcc instruction that saves memory and execution time.

If the previous jump was not taken (i.e., B > C), this instruction sets BL to 1, indicating that the condition (B > C) is true. Now, BL holds the result of (B > C).

This is the target label where the program continues after the comparison between B and C. If B <= C, the program jumps here and continues with the next set of operations.

Intel x86 and ARM Operation Types

d. Now consider the high-level language statement:

A: = (B > C) OR (D = F)

A compiler might generate the following code:

```
        MOV     EAX, B      ;move contents of location B to register EAX
        CMP     EAX, C      ;compare contents of register EAX and location C
        MOV     BL, 0       ;0 represents false
        JLE     N1          ;jump if (B ≤ C)
        MOV     BL, 1       ;1 represents false
N1      MOV     EAX, D
        CMP     EAX, F
        MOV     BH, 0
        JNE     N2
        MOV     BH, 1
N2      OR      BL, BH
```

Show an alternative implementation using the SETcc instruction that saves memory and execution time.

compares the contents of EAX (which now holds the value of D) with the value in memory location F. The comparison sets the processor flags:

If D == F, the equal flag will be set.
If D != F, the not equal flag will be set

sets the BH register to 0, representing false. We will use BH to store the result of the comparison (D == F)

Intel x86 and ARM Operation Types

d. Now consider the high-level language statement:

A: = (B > C) OR (D = F)

A compiler might generate the following code:

```
        MOV     EAX, B      ;move contents of location B to register EAX
        CMP     EAX, C      ;compare contents of register EAX and location C
        MOV     BL, 0       ;0 represents false
        JLE     N1          ;jump if (B ≤ C)
        MOV     BL, 1       ;1 represents false
N1      MOV     EAX, D
        CMP     EAX, F
        MOV     BH, 0
        JNE     N2
        MOV     BH, 1
N2      OR      BL, BH
```

Show an alternative implementation using the SETcc instruction that saves memory and execution time.

This is a conditional jump instruction that checks the comparison flags set by the CMP instruction.
•If D != F, the JNE (Jump if Not Equal) condition is true, and the program jumps to label N2.
•If D == F, the jump is not taken, and the next instruction (MOV BH, 1) is executed

If the previous jump was not taken (i.e., D == F), this instruction sets BH to 1, indicating that the condition (D == F) is true. Now, BH holds the result of (D == F

Intel x86 and ARM Operation Types

d. Now consider the high-level language statement:

A: = (B > C) OR (D = F)

A compiler might generate the following code:

```
        MOV     EAX, B      ;move contents of location B to register EAX
        CMP     EAX, C      ;compare contents of register EAX and location C
        MOV     BL, 0       ;0 represents false
        JLE     N1          ;jump if (B ≤ C)
        MOV     BL, 1       ;1 represents false
N1      MOV     EAX, D
        CMP     EAX, F
        MOV     BH, 0
        JNE     N2
        MOV     BH, 1
N2      OR      BL, BH
```

Show an alternative implementation using the SETcc instruction that saves memory and execution time.

performs a bitwise OR between the values in BL and BH. The result of the OR operation will be:
•1 if either BL or BH is 1 (i.e., if either (B > C) or (D == F) is true).
•0 if both BL and BH are 0 (i.e., if both (B <= C) and (D != F) are true).
Result of OR operation will be stored in BL. Represents the final value of the expression (B > C) OR (D = F).

This is the target label where the program continues after the comparison between D and F. If D != F, the program jumps here and continues with the next operation.

Implementation using SET CC Instructions

$A := (B > C)\ OR\ (D = F)$

We can simplify the code by directly setting the BL and BH registers based on the comparisons using the SETcc instruction, where cc is the condition code for the comparison

```
MOV EAX, B          ; Move B into EAX
CMP EAX, C          ; Compare B and C
SETG BL             ; Set BL to 1 if B > C, else set it to 0

MOV EAX, D          ; Move D into EAX
CMP EAX, F          ; Compare D and F
SETE BH             ; Set BH to 1 if D == F, else set it to 0

OR BL, BH           ; Perform logical OR between BL and BH
MOV A, BL           ; Store the result (either 0 or 1) in A
```

Reduced Memory Usage: *No need to use the extra MOV instructions to set BL and BH. SETcc instruction directly sets the flags based on the comparison result.*

Reduced Execution Time: *Eliminate the jumps (JLE, JNE) and directly set the bytes using SETcc, which is more efficient.*

No Need for Jump Logic: *no need for conditional jumps by using the SETcc instructions, which directly store result of comparison into the registers.*

Intel x86 and ARM Operation Types

12.12 The x86 architecture includes an instruction called Decimal Adjust after Addition (DAA). DAA performs the following sequence of instructions:

```
if ((AL AND 0FH) >9) OR (AF = 1)   then
      AL ← AL + 6;
      AF ← 1;
else
      AF ← 0;
endif;
if (AL > 9FH) OR (CF = 1)   then
      AL ← AL + 60H;
      CF ← 1;
else
      CF ← 0;
endif.
```

"H" indicates hexadecimal. AL is an 8-bit register that holds the result of addition of two unsigned 8-bit integers. AF is a flag set if there is a carry from bit 3 to bit 4 in the result of an addition. CF is a flag set if there is a carry from bit 7 to bit 8. Explain the function performed by the DAA instruction.

AL register: *An 8-bit register that holds the result of an addition.*

AF (Auxiliary Carry Flag): *This flag is set if there is a carry from bit 3 to bit 4 during the addition (useful for BCD arithmetic).*

CF (Carry Flag): *This flag is set if there is a carry from bit 7 to bit 8 during the addition (normal carry in binary addition).*

BCD (Binary Coded Decimal): *In BCD, each nibble (4 bits) of a byte represents a decimal digit (0-9). For example, 0x25 represents the decimal number 25*

Intel x86 and ARM Operation Types

12.12 The x86 architecture includes an instruction called Decimal Adjust after Addition (DAA). DAA performs the following sequence of instructions:

```
if ((AL AND 0FH) >9) OR (AF = 1)   then
      AL ← AL + 6;
      AF ← 1;
else
      AF ← 0;
endif;
if (AL > 9FH) OR (CF = 1)   then
      AL ← AL + 60H;
      CF ← 1;
else
      CF ← 0;
endif.
```

If (AL AND 0FH) > 9 lower nibble of AL is greater than 9, which is invalid for BCD) OR if the AF flag is set (indicating a carry from bit 3 to bit 4 during the addition), then we need to adjust the result.

If the condition is true, AL is increased by 6 (AL = AL + 6), and the AF (Auxiliary Carry Flag) is set to 1

"H" indicates hexadecimal. AL is an 8-bit register that holds the result of addition of two unsigned 8-bit integers. AF is a flag set if there is a carry from bit 3 to bit 4 in the result of an addition. CF is a flag set if there is a carry from bit 7 to bit 8. Explain the function performed by the DAA instruction.

Intel x86 and ARM Operation Types

12.12 The x86 architecture includes an instruction called Decimal Adjust after Addition (DAA). DAA performs the following sequence of instructions:

```
if ((AL AND 0FH) >9) OR (AF = 1)   then
      AL ← AL + 6;
      AF ← 1;
else
      AF ← 0;
endif;
if (AL > 9FH) OR (CF = 1)   then
      AL ← AL + 60H;
      CF ← 1;
else
      CF ← 0;
endif.
```

If (AL > 0x9F) (the value in AL is greater than 0x9F, which is invalid in BCD, since 0x9F represents the decimal value 159 which isn't a valid BCD representation) OR if the CF (Carry Flag) is set (indicating a carry from bit 7 to bit 8 during the addition), then we need to adjust the result.

"H" indicates hexadecimal. AL is an 8-bit register that holds the result of addition of two unsigned 8-bit integers. AF is a flag set if there is a carry from bit 3 to bit 4 in the result of an addition. CF is a flag set if there is a carry from bit 7 to bit 8. Explain the function performed by the DAA instruction.

Intel x86 and ARM Operation Types

12.12 The x86 architecture includes an instruction called Decimal Adjust after Addition (DAA). DAA performs the following sequence of instructions:

```
if ((AL AND 0FH) >9) OR (AF = 1)  then
      AL ← AL + 6;
      AF ← 1;
else
      AF ← 0;
endif;
if (AL > 9FH) OR (CF = 1)  then
      AL ← AL + 60H;
      CF ← 1;
else
      CF ← 0;
endif.
```

Adding 0x60 (which is 96 in decimal) shifts the value back into the correct range for BCD representation.

"H" indicates hexadecimal. AL is an 8-bit register that holds the result of addition of two unsigned 8-bit integers. AF is a flag set if there is a carry from bit 3 to bit 4 in the result of an addition. CF is a flag set if there is a carry from bit 7 to bit 8. Explain the function performed by the DAA instruction.

Intel x86 and ARM Operation Types

12.12 The x86 architecture includes an instruction called Decimal Adjust after Addition (DAA). DAA performs the following sequence of instructions:

```
if ((AL AND 0FH) >9) OR (AF = 1)  then
      AL ← AL + 6;
      AF ← 1;
else
      AF ← 0;
endif;
if (AL > 9FH) OR (CF = 1)  then
      AL ← AL + 60H;
      CF ← 1;
else
      CF ← 0;
endif.
```

"H" indicates hexadecimal. AL is an 8-bit register that holds the result of addition of two unsigned 8-bit integers. AF is a flag set if there is a carry from bit 3 to bit 4 in the result of an addition. CF is a flag set if there is a carry from bit 7 to bit 8. Explain the function performed by the DAA instruction.

DAA (Decimal Adjust after Addition) instruction is used in the x86 architecture to adjust the contents of the AL register after adding two unsigned 8-bit integers in a way that the result is converted from binary to BCD (Binary Coded Decimal).

DAA instruction adjusts the AL register so that it contains a valid BCD value.

Instruction Set Architecture: Addressing Modes and Formats

Addressing Modes

The address field in a instruction format are relatively small. Need to reference a large range of locations in main memory or virtual memory. To achieve this objective, a variety of addressing techniques has been employed. Most common addressing techniques, or modes:

- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement
- Stack

Addressing Modes

These modes are illustrated in Figure 13.1. we use the following notation:

A = contents of an address field in the instruction

R = contents of an address field in the instruction that refers to a register

EA = actual (effective) address of the location containing the referenced operand

(X) = contents of memory location X or register X

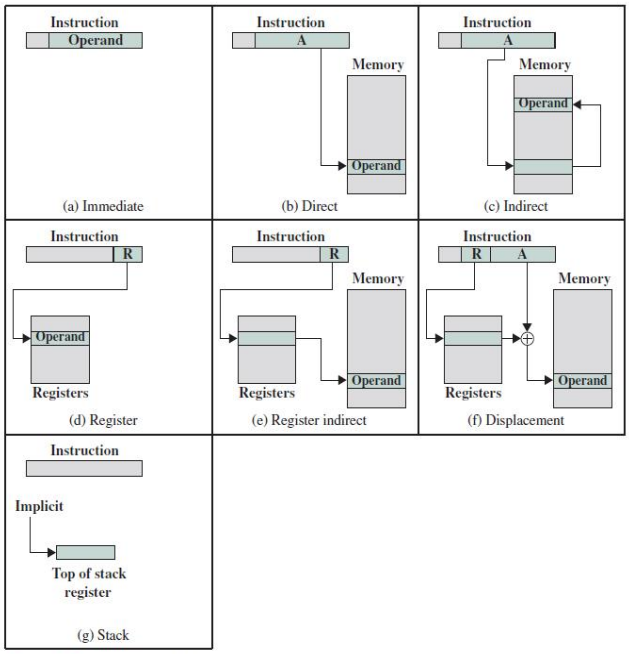


Figure 13.1 Addressing Modes

Addressing Modes

Table 13.1 indicates the address calculation performed for each addressing mode.

Table 13.1 Basic Addressing Modes

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

Processor Registers

There are ten 32-bit and six 16-bit processor registers in IA-32 architecture. The registers are grouped into three categories –

- General registers,
- Control registers, and
- Segment registers.

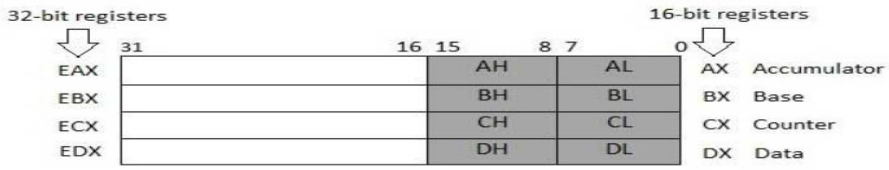
The general registers are further divided into the following groups –

- Data registers,
- Pointer registers, and
- Index registers.

Data Registers

Four 32-bit data registers are used for arithmetic, logical, and other operations. These 32-bit registers can be used in three ways –

- As complete 32-bit data registers: EAX, EBX, ECX, EDX.
- Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
- Lower and higher halves of the above-mentioned four 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH, and DL.



Some of these data registers have specific use in arithmetical operations.

AX is the primary accumulator; it is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.

BX is known as the base register, as it could be used in indexed addressing.

CX is known as the count register, as the ECX, CX registers store the loop count in iterative operations.

DX is known as the data register. It is also used in input/output operations. It is also used with AX register along with DX for multiply and divide operations involving large values.

Pointer Registers

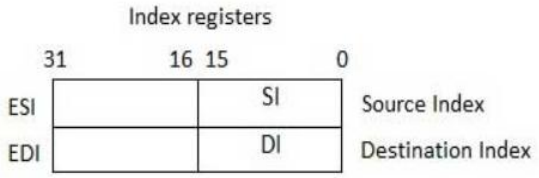
The pointer registers are 32-bit EIP, ESP, and EBP registers and corresponding 16-bit right portions IP, SP, and BP. There are three categories of pointer registers –

- **Instruction Pointer (IP)** – The 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.
- **Stack Pointer (SP)** – The 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.
- **Base Pointer (BP)** – The 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.

Index Registers

The 32-bit index registers, ESI and EDI, and their 16-bit rightmost portions. SI and DI, are used for indexed addressing and sometimes used in addition and subtraction. There are two sets of index pointers –

- **Source Index (SI)** – It is used as source index for string operations.
- **Destination Index (DI)** – It is used as destination index for string operations.



Control Registers

The 32-bit instruction pointer register and the 32-bit flags register combined are considered as the control registers.

Many instructions involve comparisons and mathematical calculations and change the status of the flags and some other conditional instructions test the value of these status flags to take the control flow to other location.

The common flag bits are:

- **Overflow Flag (OF)** – It indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.
- **Direction Flag (DF)** – It determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.
- **Interrupt Flag (IF)** – It determines whether the external interrupts like keyboard entry, etc., are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.
- **Trap Flag (TF)** – It allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through the execution one instruction at a time.

- **Sign Flag (SF)** – It shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.
- **Zero Flag (ZF)** – It indicates the result of an arithmetic or comparison operation. A nonzero result clears the zero flag to 0, and a zero result sets it to 1.
- **Auxiliary Carry Flag (AF)** – It contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.
- **Parity Flag (PF)** – It indicates the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits sets the parity flag to 1.
- **Carry Flag (CF)** – It contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a shift or rotate operation.

The following table indicates the position of flag bits in the 16-bit Flags register:

Flag:					O	D	I	T	S	Z		A		P		C
Bit no:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Addressing Modes

1. virtually all computer architectures provide more than one of these addressing modes. Different opcodes will use different addressing modes.

One or more bits in the instruction format can be used as a mode field. Value of mode field determines which addressing mode is to be used.

2. Interpretation of the effective address (EA). In a system without virtual memory, the effective address will be either a main memory address or a register.

MMU: In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the memory management unit (MMU) and is invisible to the programmer.

Addressing Modes

Immediate Addressing: The operand value is present in the instruction.This mode can be used to define and use constants or set initial values of variables. Number will be stored in two's complement form; the leftmost bit of the operand field is used as a sign bit.

When operand is loaded into a data register, the sign bit is extended to the left to full data word size. Immediate binary value is interpreted as an unsigned nonnegative integer.

Advantage: No memory reference is required to obtain the operand, saving one memory or cache cycle in the instruction cycle.

Disadvantage: Size of number is restricted to size of the address field.

An immediate operand has a constant value or an expression. When an instruction with two operands uses immediate addressing, the first operand may be a register or memory location, and the second operand is an immediate constant. The first operand defines the length of the data.

For example,

```
BYTE_VALUE  DB  150     ; A byte value is defined
WORD_VALUE  DW  300     ; A word value is defined
ADD  BYTE_VALUE, 65     ; An immediate operand 65 is added
MOV  AX, 45H            ; Immediate constant 45H is transferred to AX
```

Addressing Modes

Direct Memory Addressing

When operands are specified in memory addressing mode, direct access to main memory, usually to the data segment, is required. This way of addressing results in slower processing of data. To locate the exact location of data in memory, we need the segment start address, which is typically found in the DS register and an offset value. This offset value is also called **effective address**.

In direct addressing mode, the offset value is specified directly as part of the instruction, usually indicated by the variable name. The assembler calculates the offset value and maintains a symbol table, which stores the offset values of all the variables used in the program.

In direct memory addressing, one of the operands refers to a memory location and the other operand references a register.

For example,

```
ADD     BYTE_VALUE, DL  ; Adds the register in the memory location
MOV     BX, WORD_VALUE  ; Operand from the memory is added to register
```

Addressing Modes

Direct-Offset Addressing

This addressing mode uses the arithmetic operators to modify an address. For example, look at the following definitions that define tables of data –

```
BYTE_TABLE DB  14, 15, 22, 45      ; Tables of bytes
WORD_TABLE DW  134, 345, 564, 123  ; Tables of words
```

The following operations access data from the tables in the memory into registers –

```
MOV CL, BYTE_TABLE[2]   ; Gets the 3rd element of the BYTE_TABLE
MOV CL, BYTE_TABLE + 2  ; Gets the 3rd element of the BYTE_TABLE
MOV CX, WORD_TABLE[3]   ; Gets the 4th element of the WORD_TABLE
MOV CX, WORD_TABLE + 3  ; Gets the 4th element of the WORD_TABLE
```

Addressing Modes

Indirect Memory Addressing

This addressing mode utilizes the computer's ability of Segment:Offset addressing. Generally, the base registers EBX, EBP (or BX, BP) and the index registers (DI, SI), coded within square brackets for memory references, are used for this purpose.

Indirect addressing is generally used for variables containing several elements like, arrays. Starting address of the array is stored in, say, the EBX register.

The following code snippet shows how to access different elements of the variable.

```
MY_TABLE TIMES 10 DW 0  ; Allocates 10 words (2 bytes) each initialized to
MOV EBX, [MY_TABLE]     ; Effective Address of MY_TABLE in EBX
MOV [EBX], 110          ; MY_TABLE[0] = 110
ADD EBX, 2              ; EBX = EBX +2
MOV [EBX], 123          ; MY_TABLE[1] = 123
```

Addressing Modes

Register Addressing: Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address: **EA = R**

If the contents of a register address field in an instruction is 5, then register R5 is the intended address, and the operand value is contained in R5. Address field that references registers will have from 3 to 5 bits, so a total of 8 to 32 general-purpose registers can be referenced.

Advantages: (1) only a small address field is needed in the instruction, (2) no time-consuming memory references are required.

Memory access time for a register internal to the processor is much less than that for a main memory address.

Disadvantage: Address space is very limited.

If register addressing is heavily used in an instruction set, this implies that the processor registers will be heavily used.

If every operand is brought into a register from main memory, operated on once and returned to main memory its a wasteful. If operand in a register remains in use for multiple operations, then a real savings is achieved

Addressing Modes

Programmer or compiler decide which values should remain in registers and which should be stored in main memory. Modern processors employ multiple general-purpose registers, placing a burden for efficient execution on the assembly-language programmer (e.g., compiler writer).

Register Indirect Addressing: Register indirect addressing is analogous to indirect addressing. Only difference is address field refers to a memory location or a register. For register indirect address,

EA = (R)

Advantages and limitations of register indirect addressing are basically the same as for indirect addressing.

Address space limitation (limited range of addresses) of the address field is overcome by having that field refer to a word-length location containing an address.

Register indirect addressing uses one less memory reference than indirect addressing.

Addressing Modes

Displacement Addressing: A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing. It is known by a variety of names depending on the context of its use, but the basic mechanism is the same. We will refer to this as displacement addressing:

EA = A + (R)

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address .

Three of the most common uses of displacement addressing:

1. Relative addressing
2. Base-register addressing
3. Indexing

Addressing Modes

1. Relative addressing: Also called PC-relative addressing, the implicitly referenced register is the program counter (PC). Next instruction address is added to the address field to produce EA (Effective Address).

Address field is treated as a twos complement number for this operation. Effective address is a displacement relative to the address of the instructio n.

Relative addressing exploits the concept of locality. If most memory references are relatively near to the instruction being executed, then the use of relative addressing saves address bits in the instruction.

2.Base-register addressing: Referenced register contains a main memory address, and address field contains a displacement (usually an unsigned integer representation) from that address. The register reference may be explicit or implicit.

Base-register addressing also exploits the locality of memory references. i)Single segment-base register is employed and is used implicitly.2) programmer choose a register to hold base address of a segment, and the instruction must reference it explicitly.

If the length of the address field is K and the number of possible registers is N, then one instruction can reference any one of N areas of 2^K words.

Addressing Modes

3. Indexing: Address field references a main memory address, and referenced register contains a positive displacement from that address.Indexing provide an efficient mechanism for performing iterative operations.

For example, a list of numbers stored starting at location A and add 1 to each element. We need to fetch each value, add 1 to it, and store it back. Sequence of effective addresses is A, A + 1, A + 2, . . . , up to last location on list.

With indexing,value A is stored in instruction's address field, and in index register initialized to 0. After each operation, index register is incremented by 1.

Autoindexing: *N*eed to increment or decrement index register after each reference to it. some systems will automatically do this as part ofsame instruction cycle. This is known as autoindexing. $EA = (A) + (R)$

If general- purpose registers are used, autoindex operation may need to be signaled by a bit in the instruction.

$$EA = A + (R)$$
$$(R) \leftarrow (R) + 1$$

Addressing Modes

Postindexing: Some machines have both indirect addressing and indexing and it is possible to employ both in same instruction. There are two possibilities: indexing is performed either before or after indirection. If indexing is performed after indirection, it is termed postindexing.

First, contents of the address field are used to access a memory location containing a direct address. This address is then indexed by the register value. For example, OS needs to employ a process control block for each process.

Addresses in instructions that reference the block could point to a location (value = A) containing a variable pointer to the start of a process control block. Index register contains the displacement within the block.With preindexing, indexing is performed before the indirection:

$$EA = (A + (R))$$

An address is calculated as with simple indexing. Calculated address contains not the operand, but the address of the operand.

Example is to construct a multiway branch table. Table of addresses can be set up starting at location A. By indexing into this table, the required location can be found.

Addressing Modes

Stack Addressing: Stack is a linear array of locations and work as last-in-first-out queue. The stack is a reserved block of locations.

Items are appended to the top of the stack, at any given time block is partially filled. Associated with the stack is a pointer whose value is the address of top of the stack.

Top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack.

The stack pointer is maintained in a register. References to stack locations in memory are in fact register indirect addresses.

x86 Addressing Modes

x86 address translation mechanism produces an address, called a virtual or effective address, that is an offset into a segment.

Segmentation is the process in which the main memory of the computer is logically divided into different segments and each segment has its own base address. It is basically used to enhance the speed of execution of the computer system, so that the processor is able to fetch and execute the data from the memory easily and fast.

Sum of the starting address of the segment and the effective address produces a linear address.

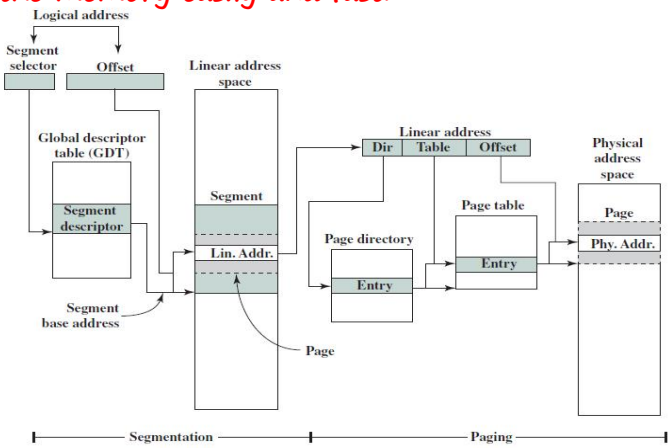


Figure 8.21 Intel x86 Memory Address Translation Mechanisms

x86 and ARM Addressing Modes

The process of retrieving processes in the form of pages from the secondary storage into the main memory is known as paging. The basic purpose of paging is to separate each procedure into pages.

If paging is being used, this linear address must pass through a page-translation mechanism to produce a physical address.

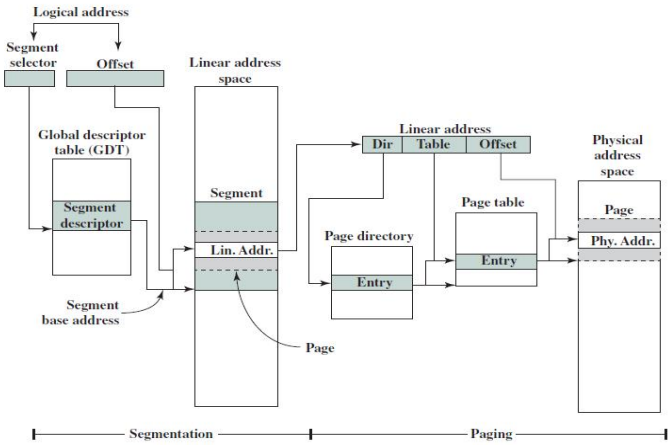


Figure 8.21 Intel x86 Memory Address Translation Mechanisms

x86 and ARM Addressing Modes

Figure 13.2 indicates six segment registers; the one being used for a particular reference depends on the context of execution and the instruction.

Code segment register (CS): *is used for addressing memory location in the code segment of the memory, where the executable program is stored.*

Data segment register (DS): *points to the data segment of the memory where the data is stored.*

Extra Segment Register (ES): *also refers to a segment in the memory which is another data segment in the memory.*

Stack Segment Register (SS): *is used for addressing stack segment of the memory. It is that segment of memory which is used to store stack data.*

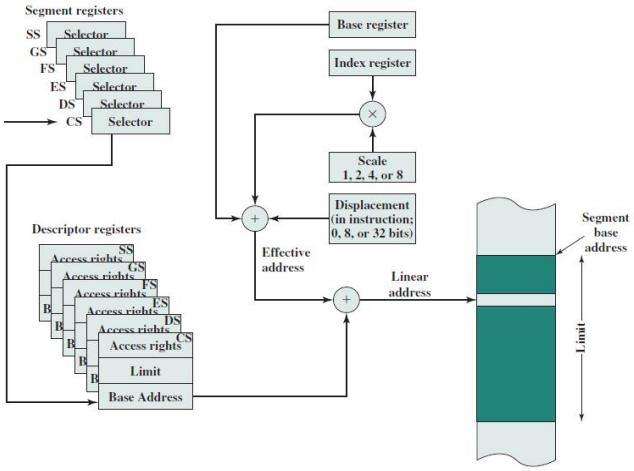
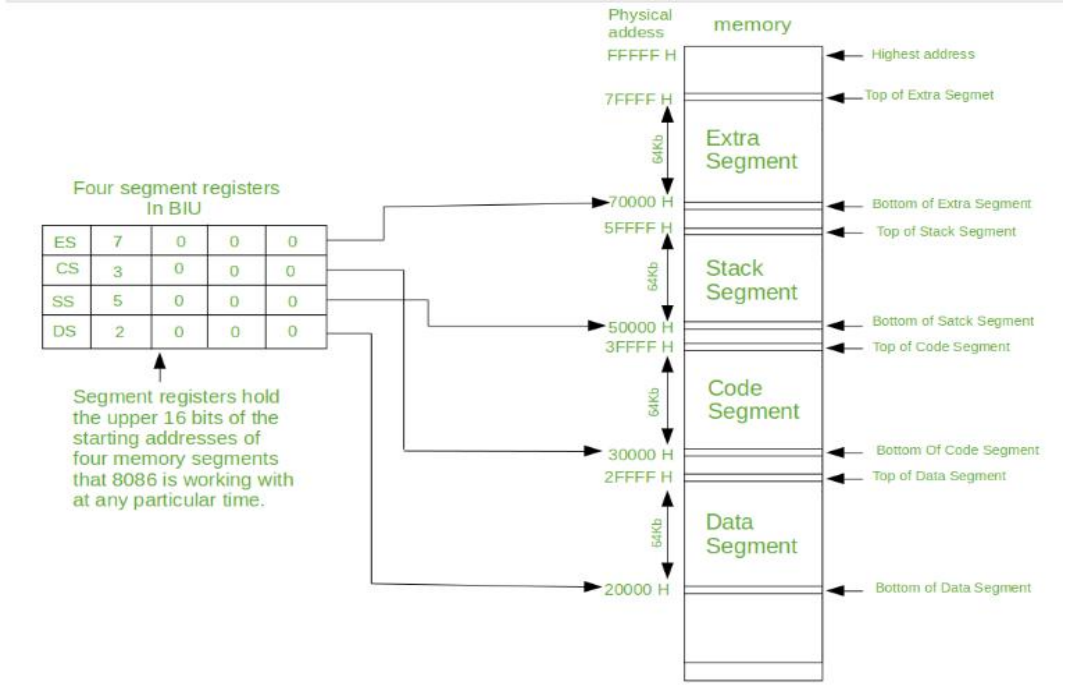


Figure 13.2 x86 Addressing Mode Calculation

x86 and ARM Addressing Modes



x86 and ARM Addressing Modes

Each segment register holds an index into the segment descriptor table, which holds the starting address of the corresponding segments.

Associated with each user visible segment register is a segment descriptor register, which records the access rights for the segment as well as the starting address and limit (length) of the segment.

There are two registers that may be used in constructing an address: base register and index register.

Base Register(BX): used to hold the base address of memory location for reading and writing data into the memory.

Index register: used for pointing to operand addresses during the run of a program. Also used for holding loop iterations and counters. In some architectures, it is used for read/writing blocks of memory.

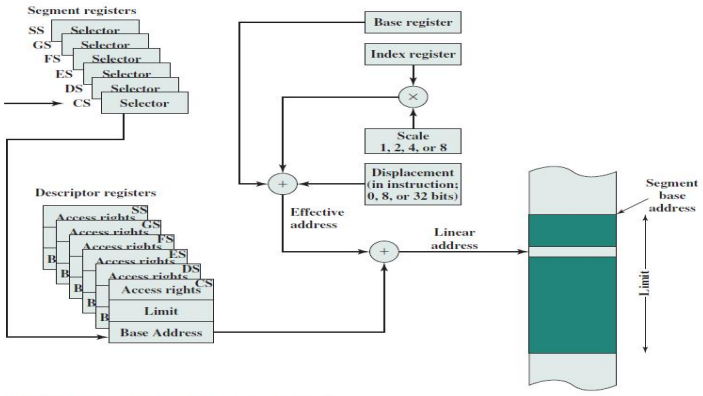


Figure 13.2 x86 Addressing Mode Calculation

x86 Addressing Modes

Table 13.2 lists the x86 addressing modes.

Table 13.2 x86 Addressing Modes

Mode	Algorithm
Immediate	Operand = A
Register Operand	LA = R
Displacement	LA = (SR) + A
Base	LA = (SR) + (B)
Base with Displacement	LA = (SR) + (B) + A
Scaled Index with Displacement	LA = (SR) + (I) × S + A
Base with Index and Displacement	LA = (SR) + (B) + (I) + A
Base with Scaled Index and Displacement	LA = (SR) + (I) × S + (B) + A
Relative	LA = (PC) + A

LA = linear address
(X) = contents of X
SR = segment register
PC = program counter
A = contents of an address field in the instruction
R = register
B = base register
I = index register
S = scaling factor

x86 Addressing Modes

For **immediate mode**, operand is included in the instruction. The operand can be a byte, word, or doubleword of data.

For **register operand mode**, the operand is located in a register.

For general instructions, such as data transfer, arithmetic, and logical instructions, the operand can be one of the 32-bit general registers: (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP), one of the 16-bit general registers (AX, BX, CX, DX, SI, DI, SP, BP), or one of the 8-bit general registers (AH, BH, CH, DH, AL, BL, CL, DL).

x86 Addressing Modes

Displacement mode: the operand's offset is contained as part of the instruction as an 8-, 16-, or 32-bit displacement.

Displacement value can be of 32 bits, making for a 6-byte instruction. Displacement addressing can be useful for referencing global variables.

The remaining addressing modes are indirect as address portion of instruction tells processor where to look to find the address.

Base mode: specifies that one of the 8-, 16-, or 32-bit registers contains the effective address. This is equivalent to register indirect addressing.

Base with displacement mode: instruction includes a displacement to be added to a base register, which may be any of the general-purpose registers.Examples of uses of this mode are as follows:

- ✓ Used to point to the start of a local variable area. For example, the base register could point to beginning of stack frame which contains local variables.
- ✓ Used to index an array when element size is not 1, 2, 4, or 8 bytes and cannot be indexed using an index register. Displacement points to beginning of array and base register holds results to determine offset to specific element within the array.

x86 Addressing Modes

In **scaled index with displacement mode:** instruction includes a displacement to be added to a register, called an index register. In calculating the effective address, contents of index register are multiplied by a scaling factor of 1, 2, 4, or 8, and then added to a displacement. This mode is very convenient for indexing arrays.

- ✓ A scaling factor of 2 can be used for an array of 16-bit integers.
- ✓ A scaling factor of 4 can be used for 32-bit integers or floating-point numbers.
- ✓ A scaling factor of 8 can be used for an array of double-precision floating-point numbers.

Base with index and displacement mode: sums the contents of the base register, index register, and a displacement to form the effective address.

Base register can be any general-purpose register and the index register can be any general- purpose register except ESP.Example:

- ✓ Used for accessing a local array on a stack frame.
- ✓ Used to support a two-dimensional array,displacement points to beginning of array and each register handles one dimension of the array.

x86 Addressing Modes

Scaled index with displacement mode: sums contents of index register multiplied by a scaling factor, contents of base register, and displacement.

- ✓ Useful if an array is stored in a stack frame; array elements would be 2, 4, or 8 bytes each in length.
- ✓ Provides efficient indexing of a two- dimensional array when the array elements are 2, 4, or 8 bytes in length.

Relative addressing: can be used in transfer- of-control instructions. A displacement is added to the value of the program counter, which points to the next instruction.

Displacement is treated as a signed byte, word, or doubleword value, and that value either increases or decreases the address in the program counter.

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4]←Regs[R4]+Regs[R3]`	When a value is in a register
Immediate	`Add R4,3`	`Regs[R4]←Regs[R4]+3`	For constants
Displacement	`Add R4,100(R1)`	`Regs[R4]←Regs[R4]+Mem[100+Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	`Add R4,(R1)`	`Regs[R4]←Regs[R4]+Mem[Regs[R1]]`	Accessing using a pointer or a computed address
Indexed	`Add R3,(R1+R2)`	`Regs[R3]←Regs[R3]+Mem[Regs[R1]+Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount
Direct or absolute	`Add R1,(1001)`	`Regs[R1]←Regs[R1]+Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large
Memory indirect	`Add R1,@(R3)`	`Regs[R1]←Regs[R1]+Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer *p*, then mode yields **p*
Autoincrement	`Add R1,(R2)+`	`Regs[R1]←Regs[R1]+Mem[Regs[R2]]` `Regs[R2]←Regs[R2]+d`	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*
Autodecrement	`Add R1,-(R2)`	`Regs[R2]←Regs[R2]-d` `Regs[R1]←Regs[R1]+Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1]←Regs[R1]+Mem[100+Regs[R2]+Regs[R3]*d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers

Figure A.6 Selection of addressing modes with examples, meaning, and usage. In autoincrement/-decrement and scaled addressing modes, the variable *d* designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes). These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use displacement addressing to simulate register indirect with 0 for the address and to simulate direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode. The extensions to C used as hardware descriptions are defined on page A.38.