Indirect Addressing Modes

A (more complete) Summary of Intel Addressing Modes

Immediate: constant

- constant data contained as *part of instruction*

Register: contained in a register

- Instruction specifies which *register*

Memory: contained in a memory location

- Instruction specifies *address of memory* operand

Direct

- constant address contained as part of the instruction
- Address static: known at assembly-time. Remains constant throughout program execution.

Indirect

- Variable address in a *register*
- instruction specifies the register holding the address
- Address dynamic: depends on contents of register when instruction is executed.

(Register) Indirect Memory Address Mode

- **Register** holds **address offset** of operand
- Register can only be: BX, SI, DI, BP
- DS default segment for BX, SI, DI
- SS default segment for BP (later!)
- Syntax: [register]

"[" & "]" differentiates from register mode !

- Indirect Example:

```
W       DW
  . . .
MOV     BX,   OFFSET W
  . . .
MOV     AX, [ BX ]
```

offset of variable treated as a constant

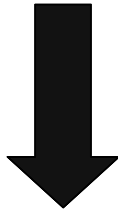
Value in BX used as offset to memory operand
- loads AX with contents of W

An Alternative to OFFSET : LEA instruction

- Before using indirect addressing, we always need to
  load the register with the address

- Two alternatives :

```
MOV   BX,  OFFSET W
```



Functionally equivalent!

```
LEA   BX, W
```

- Both calculate and load the 16-bit effective address of a memory operand.

Why are the [ ] – brackets needed?

- What is the difference?

```
MOV   AX, 1234h
MOV   AX, [ 1234h ]
```

- What is the difference?

```
MOV   AX, BX
MOV   AX, [ BX ]
```

- What is the difference? (suppose BX = 1234h)

```
MOV   AX, [1234h]
MOV   AX, [ BX ]
```

Example : Using Register Indirect **for Array Programming**

Each element is 2 bytes long

- Suppose we have array of integers declared:

```
X           DW              ; 1st element of array
            DW              ; 2nd element of array
            . . .                  etc.
numX        DW              ; number of elements in X
```

- Write a program that sums the contents of the array into AX

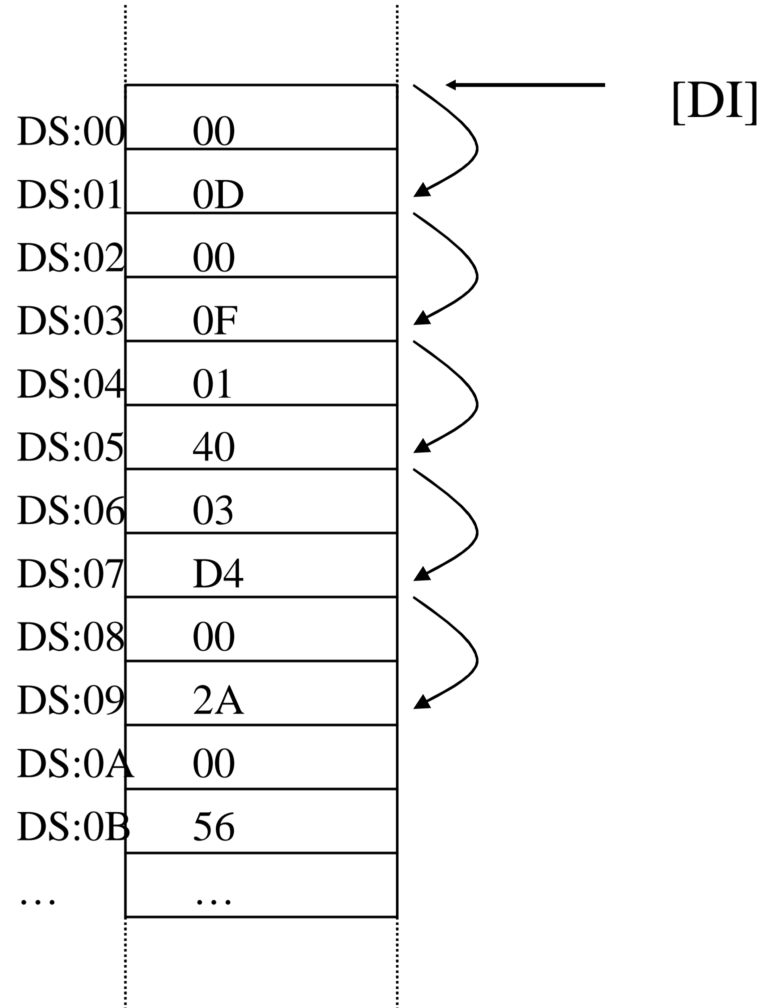
```
 int total = 0;
 for (int i=0; i<numX; i++) {
     total += X[i];
 }
```

- Use DI to hold the address of the current element.
  - i.e., it plays the role of X[i]

AX = total

CX = loop counter *i*

DI = address of X[i]



Code Fragment Example:

```
        MOV   AX, 0        ; initialize sum
        LEA   DI, X        ; initialize array index
        MOV   CX, numX     ; get # of elements

CheckForDone:
        CMP   CX, 0        ; any elements left to sum?
        JE    Done

        ADD   AX, [ DI ]   ; sum ith element

        ADD   DI, 2        ; adjust index (offset)
        SUB   CX, 1        ; one less element
        JMP   CheckForDone

Done:   …
```

Why “2”?

Segment Override

Recall : **DS** is default segment for: **BX**, **SI**, **DI**
SS is default segment for **BP** (later!)

```
MOV [BX], 3 = =   MOV DS:[BX], 3
MOV [BP], 3 = =   MOV SS:[BP], 3
```

You run out of registers: need to use either the index registers or segment registers outside of their default roles (eg. duplicating data structures),

```
MOV SS:[BX], 3
MOV ES:[BX], 3
MOV DS:[BP], 3
```

Indirect Addressing Ambiguity

- Indirect addressing: register holds offset to operand
  - Address offset points to a single memory location

- In some cases, **no** ambiguity in operand size

  eg. `MOV AL, [BX]`
    - AL – 8-bit register
    - [BX] is interpreted as the offset to byte of memory

- In others, ambiguity :

  eg `MOV [BX], 1`
  - Source is immediate value
  - 8 bit or 16 bit ? Move byte 01 to the offset, or the word 0001?

- You must use syntax qualifiers to remove ambiguity

```
MOV     BYTE PTR [BX], 1
MOV     WORD PTR [BX], 1
```

The Power of Indirect Addressing

- Advantage ?

```
MOV AX, var          versus          LEA BX, var
                                     MOV AX, [BX]
```

- Power: support for data structures
  - Arrays: collection of elements all of the same type

```
int array[10];
array[0] = 1; array[2] = 5; …
```

High level language selector is **[ ]**

  - Records or Structures: collection of elements of different types

```
struct {
      char  name[80];
      int number;
} student;
student.number = 123456;
```

High level language selector is “**.**”

Arrays are Indexable Data Structures

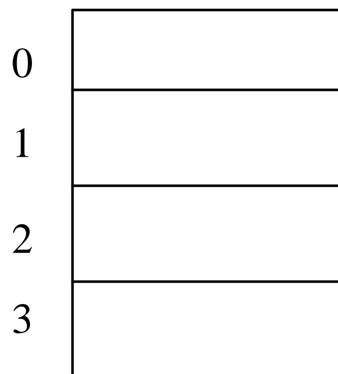
SI and DI are **index** registers. Hmmm …

Access to each element given by an address calculation

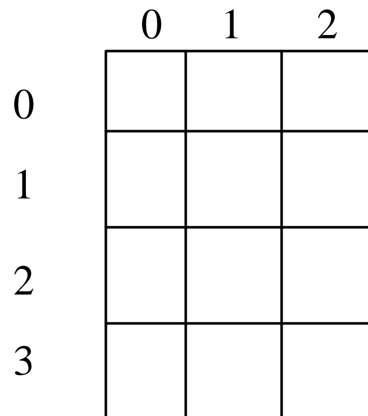
- array[i] = **start address** of the array + **i * sizeOfElement**
- Offset **depends on size of the element**

- First element of a vector is associated with the index 0. Why ?

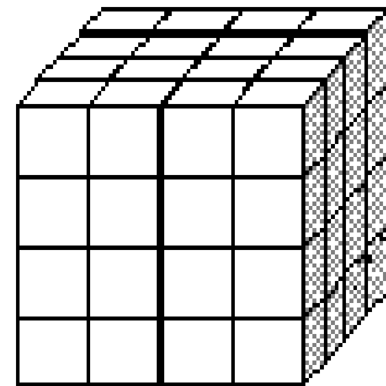
A Vector



A 2-D Matrix (4x3)



A 3-D Matrix (4x4x4)



ASM Programming of Arrays is all about Addresses

```
.data
a1 db 11h, 22h, 33h, 44h, 55h
a2 db 01h
   db 02h
   db 03h

a3 db  2 dup(0FFh)
```

Memory Map

Address	Value
DS:00	11h
DS:01	22h
DS:02	33h
DS:03	44h
DS:04	55h
DS:05	01h
DS:06	02h
DS:07	03h
DS:08	FFh
DS:09	FFh
DS:0A	?
DS:0B	?

C equivalent :

```
byte a1[5]; a1[0] = 17;
byte a2[3]; a2[1] = 2;
byte a3[2]; a3[1] = 255;
```

ASM Programming of Arrays is all about Addresses

```
  .data
a1   dw 11h, 22h, 33h, 44h
a2   dw 01h
     dw 02h
     dw 03h
a3   dw 2 dup(0FFFFh)
```

C equivalent :

```
int a1[5];   a1[0] = 17;
int a2[3];   a2[1] = 2;
int a3[2];   a3[1] = 65535;
```

Memory Map

Address	Value
DS:00	11
DS:01	00
DS:02	22
DS:03	00
DS:04	33
DS:05	00
DS:06	44
DS:07	00
DS:08	01
DS:09	00
DS:0A	02
DS:0B	00
DS:0C	03
	…

ASM Programming of Arrays is all about Addresses

```
  .data
a1   db 01h, 05h
     db 02h, 06h
     db 03h, 07h
a2   db 2*2 dup(00)
```

Memory Map

Address	Value
DS:00	01
DS:01	05
DS:02	02
DS:03	06
DS:04	03
DS:05	07
DS:06	00
DS:07	00
DS:08	00
DS:09	00
DS:0A	??
DS:0B	??
DS:0C	??

C equivalent :

```
byte a1[3][2]; a1[0][1] = 5;
byte a2[2][2]; a2[1][1] = 0;
```

Structures

• Group of related variables accessed through a common name.
• Each item within a structure: its own data type.

```
struct catalog_tag {
    char            author [40];
    char            title [40];
    char            pub [40];
    unsigned int    date;
    unsigned char   rev;
} card;
```

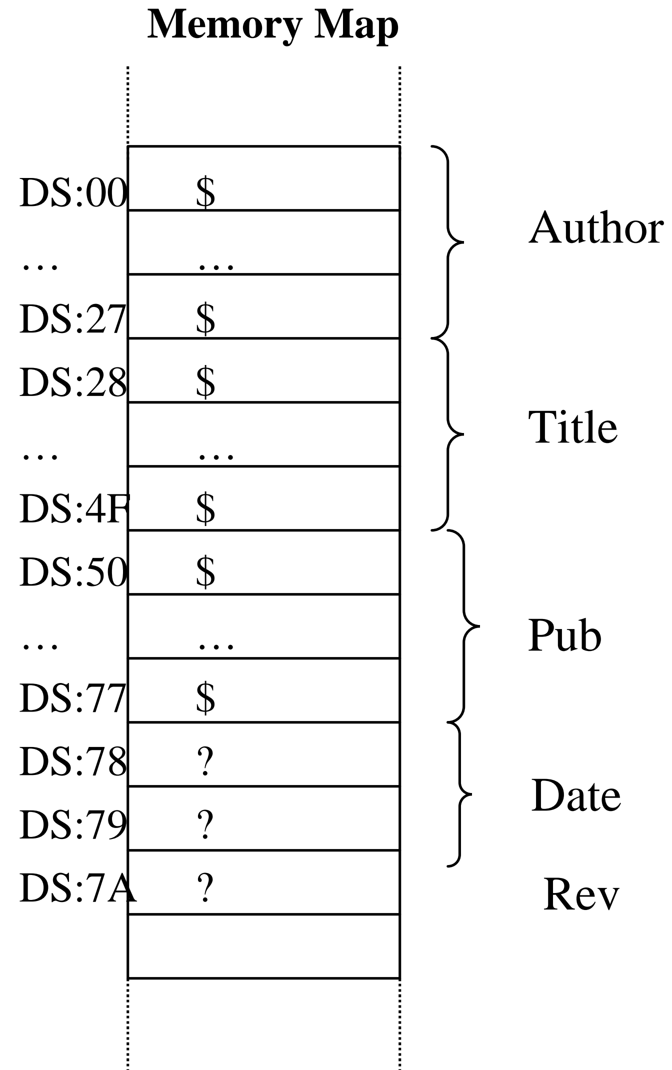
To access :

```
card.author[0]
card.date
card.rev
```

where, the variable *card* is of type *catalog_tag*.

ASM Programming of Structures is all about Addresses

```
.data
card  db 40 dup ('$')
      db 40 dup ('$')
      db 40 dup ('$')
      dw ?
      db  ?
```



There are 2 scenarios when programming arrays:

1. Address of array *static*
2. Address of array *dynamic*
   - Code work for any array of a given type (i.e. sizeof element), given the start address.
   - Example: function accepting array as an argument for processing
     - **Different invocations** of the function may process **different arrays**

- 8086 addressing modes exist to support both cases !

(Indirect) Indexed or Based Addressing Mode

- Like register indirect, except you also specify a constant

  e.g. [BX + constant]
- **During execution**, processor uses a temporary register to calculate

  BX + constant
  - It then accesses memory addressed by BX + constant
- Restriction: may only use BP, BX, SI or DI ← same as register indirect

- Typical Uses :
  1. Accessing array using static address
     - Constant = **start address** of array
     - Register = **index offset** (address offset from start)
  2. Accessing elements within a structure
     - Register = **start address** of structure
     - Constant = **offset of elements** within structure

Code Fragment Example:

```
        MOV    AX, 0            ; initialize sum
        MOV    DI, 0            ; initialize index offset
        MOV    CX, numX         ; get # of elements

CheckForDone:
        CMP    CX, 0 ; any elements left to sum?
        JE     Done

        ADD    AX, [ DI + X ]      ; sum ith element

        ADD    DI, 2            ; adjust index (offset)
        SUB    CX, 1            ; one less element
        JMP    CheckForDone

Done: …
```

Example: using indexed mode on a structure

Assume we have the previous structure definition :

```
struct catalog_tag {
            char  author [40];
            char  title [40];
            char  pub [40];
            unsigned int date;
            unsigned char rev;
      } card;
```

Write a code fragment to clear all fields of the structure

- Make all strings "empty"
- Make all data fields null.

Structure Example:

```
AUTHOR      EQU 0
TITLE       EQU AUTHOR+40
PUBLISHER   EQU TITLE+40
DATE        EQU PUBLISHER+40
REVISION    EQU DATE+2

MOV   BX, offset card
```

BX = **constant** start address of structure.

constant is the offset of element within structure.

```
MOV   BYTE PTR [BX+AUTHOR], '$'
MOV   BYTE PTR [BX+TITLE], '$'
MOV   BYTE PTR [BX+PUBLISHER], '$'
MOV   WORD PTR [BX+DATE], 0
MOV   BYTE PTR [BX+REVISION], 0
```

(Indirect) Base-Indexed Addressing Mode:

- Like indexed, except you use a second register instead of a constant
  e.g. [BX + SI]
- **During execution**, processor uses temporary register to calculate sum of register values
  – It then accesses memory addressed by sum
- Restrictions:
  – one must be base register: BX (or BP ← later!)
  – one must be index register: SI or DI
  – only legal forms:

Default DS	[BX + SI]	[BX + DI]	**base = BX**
Default SS	[BP + SI]	[BP + DI]	**base = BP**

- Typical use : **access array using dynamic address**
  – One register = **start address**; Other register = **index offset**

Code Fragment Example:

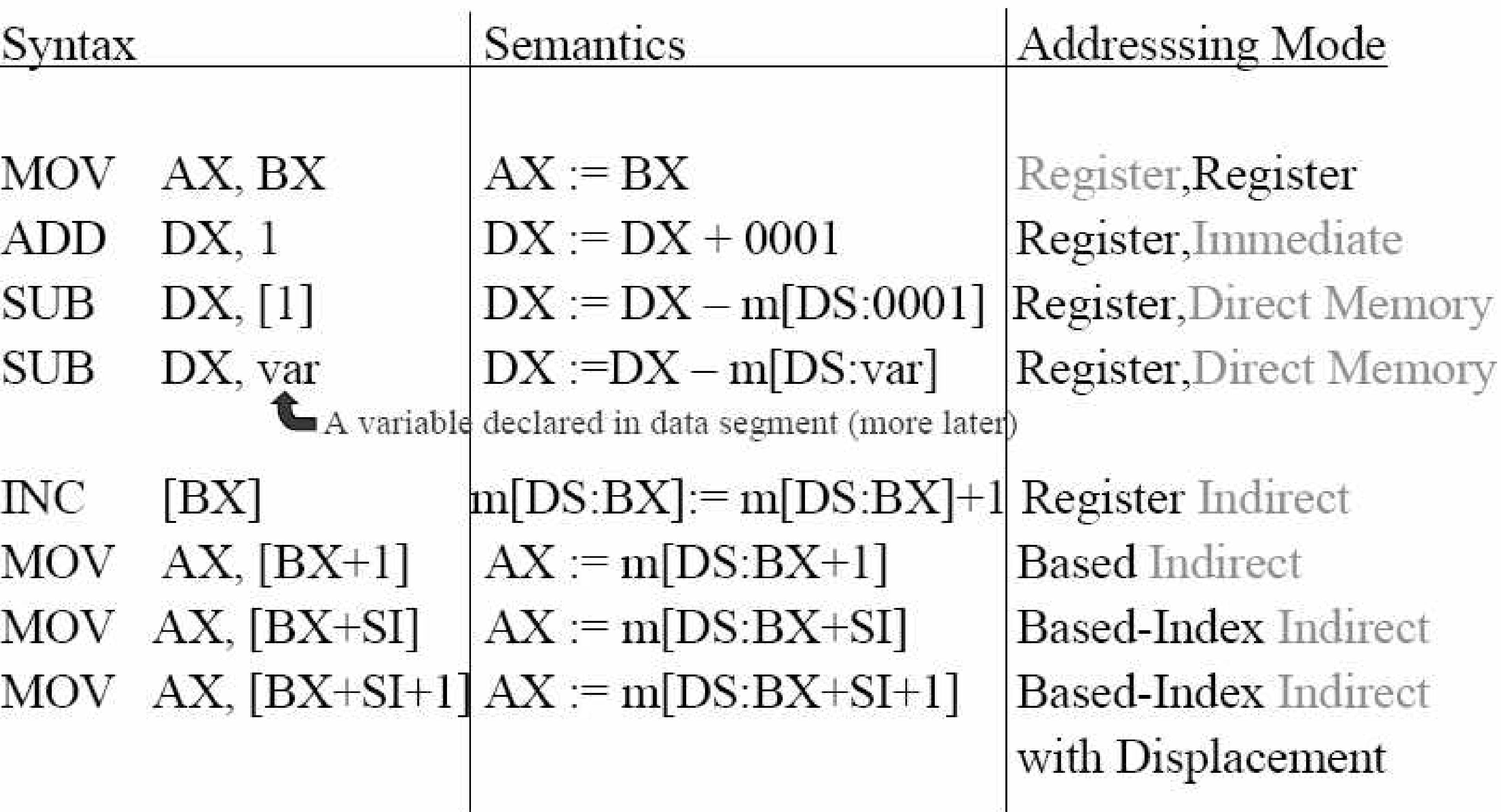
```
MOV   AX, 0        ; initialize sum
LEA   BX, X        ; initialize base
MOV   SI, 0        ; initialize index offset
MOV   CX, numX     ; get # of elements

CheckForDone:
MOV   CX, 0 ; any elements left to sum?
JE    Done

ADD   AX, [ BX + SI ]    ; sum ith element

ADD   SI, 2        ; adjust index (offset)
SUB   CX, 1        ; one less element
JMP   CheckForDone

Done: …
```

(Indirect) Based-Indexed with Displacement Addressing Mode

- Like based-indexed mode; includes a constant too

  e.g. [BX + SI + constant]
- **During execution**, processor uses temporary register to calculate sum of values
  - accesses memory addressed by sum
- Restrictions: same as based mode
- **Typical use**: **Composite Data structure** (Beyond single arrays)
  - Array of arrays :
    - Constant = **start address**,
    - 1st register = index offset into **first dimension**;
    - 2nd register = index offset into **second dimension**
  - Array of structs:
    - 1st register = **start address**;
    - 2nd register = index offset (**start of structure**);
    - constant = offset of **element within structure**.

Addressing Modes



Syntax	Semantics	Addresssing Mode
MOV AX, BX	AX := BX	Register,Register
ADD DX, 1	DX := DX + 0001	Register,Immediate
SUB DX, [1]	DX := DX – m[DS:0001]	Register,Direct Memory
SUB DX, var	DX :=DX – m[DS:var]	Register,Direct Memory
↑ A variable declared in data segment (more later)		
INC [BX]	m[DS:BX]:= m[DS:BX]+1	Register Indirect
MOV AX, [BX+1]	AX := m[DS:BX+1]	Based Indirect
MOV AX, [BX+SI]	AX := m[DS:BX+SI]	Based-Index Indirect
MOV AX, [BX+SI+1]	AX := m[DS:BX+SI+1]	Based-Index Indirect with Displacement

Using Register Indirect with Control Flow

- Three forms of addressing were identified for control flow
  - Absolute
  - Relative
  - Indirect

- Example :

```
MOV    BX, offset version1
JMP    [BX]              ; What does BX contain ?

version1:
       …
version2 :
       …
```

- Dynamic selection of code (e.g. Java runtime binding ?)

Using Register Indirect with Control Flow

- When using indirect target, you encounter another "operand ambiguity":
  Intra-segment or inter-segment control flow.

- Intra-segment control flow :
  - BX = address containing offset of next instruction.
    - BX an address to a word.
- Inter-segment control flow :
  - BX = address containing segment:offset of next instruction
    - BX is an address of two words.

- For data, we used WORD PTR and BYTE PTR to resolve the ambiguity.
- For control flow:
  - JMP NEAR PTR [BX] ; Intra-segment
  - JMP FAR PTR [BX] ; Inter-segment.

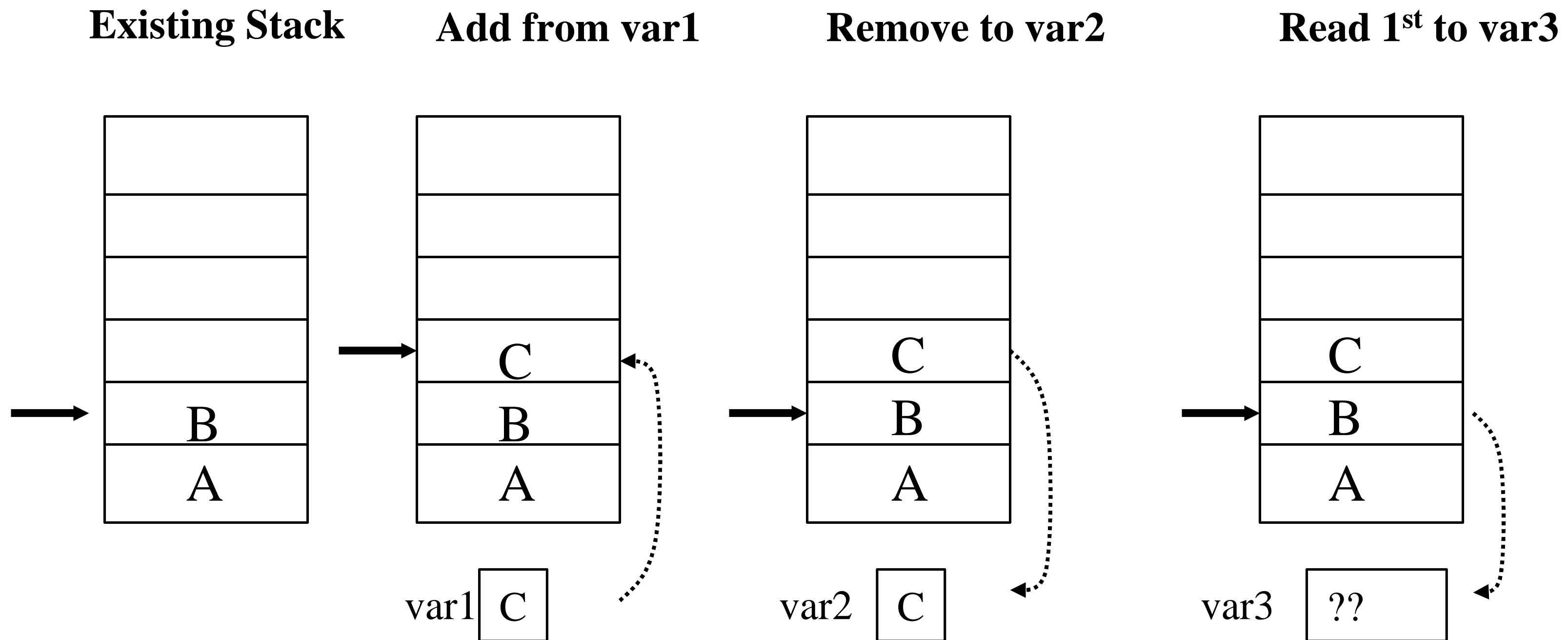
Stack … As a generic Abstract Data Type

- Stack: data structure with **LIFO behaviour** (often used to hold values temporarily)
- Concept: a pile of "things" (elements); one element on top, the rest follow beneath sequentially.
  - Example: stack of papers
  - Last-In-First-Out Behaviour :
    - Each new element **added to the top** (new thing now on top).
    - Elements removed only **from the top** (element below top in the pile is now on top).
    - Elements below top can be **looked at** if you know the position of the element relative to the top.
      - Example : Look at the 2nd element from the top.

Change State

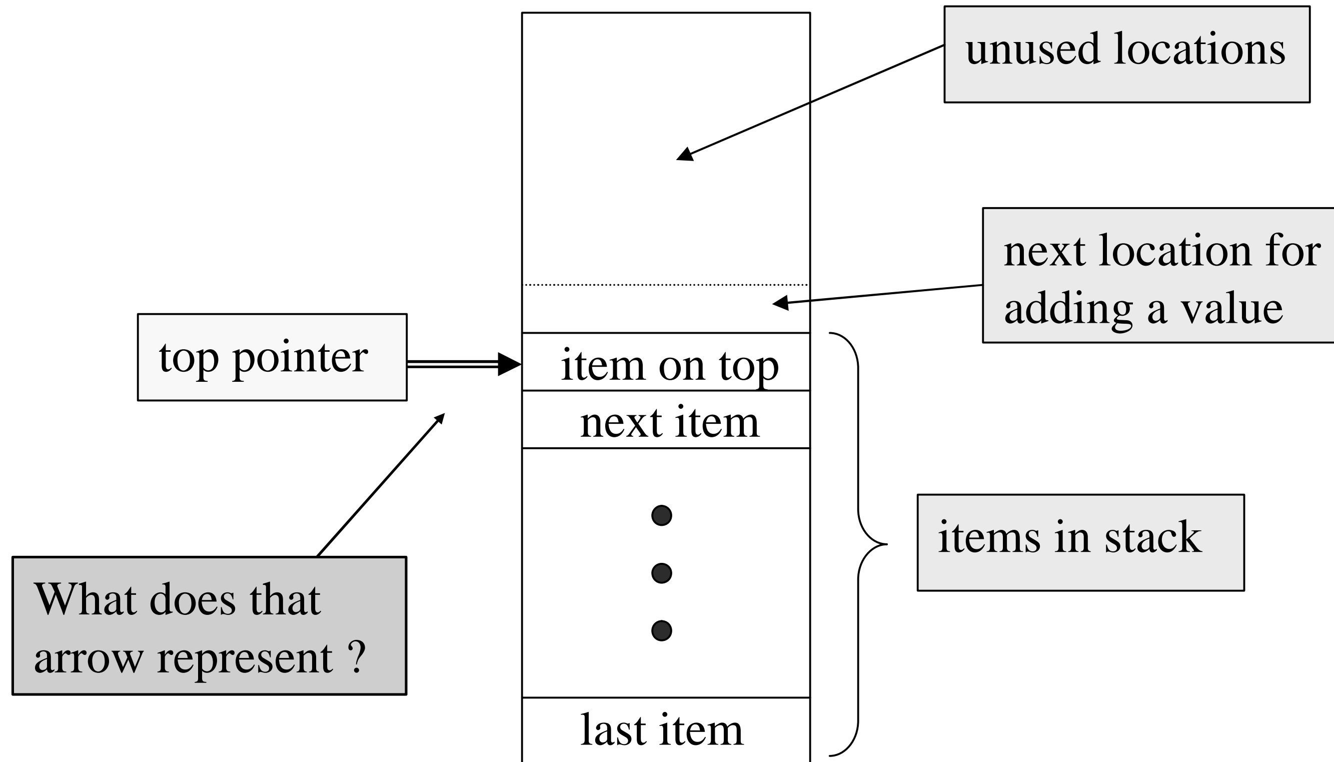
Read State

Stack … As a generic Abstract Data Type



A Generic Stack Implementation

Stack implemented by reserving (1) a block of memory to hold values and (2) a pointer to point to the value on top.



Issues in a Generic Stack Implementation

- Stack grow from high-to-low addresses (as in picture), or vice versa ?
  - Conceptually, no difference.
  - Convention: typically high-to-low
- Initialization (stack empty): what should top pointer point to?
  - usually “top” points outside the reserved block; next add will adjust pointer before copying value into the location.
- Stack overflow: full stack (no space for more items), what if item is added ?
  - How can the implementation check ?
  - What should the implementation do ?
- Stack underflow: empty stack, what if item is removed ?

A Generic Stack Implementation (Stack of bytes)

```
.data
top      dw    102h          ; Address of top element
stack   db 100h dup (?)      ; Stack = array of bytes
…
     MOV BX, top             ; Add AL to stack
     SUB BX, 1
     MOV [BX], AL
     MOV top, BX
…
     MOV BX, top             ; Read 7th element from top
     ADD BX, 7               ; into AL
     MOV AL, [BX]
…
     MOV BX, top             ; Remove top element into AL
     MOV AL, [BX]
     ADD BX, 1
     MOV top, BX
```

The Hardware or Runtime Stack

- Processor has built-in stack support called the runtime stack.
  - "top" maintained by dedicated pointer registers: SS:SP
  - SS:SP points to a stack holding 16-bit values.
  - grows "down" in memory (high-to-low addresses)
  - some instructions: use it implicitly (use/alter SS:SP)

- Must initialize SP before using any stack operations.

  .stack size

  - assembler reserved a block of memory to use for stack
  - translated into instruction to loader to initialize SS:SP !!
  - SP points at byte just above (high address!) last byte reserved for stack

Intel Stack Instructions

- **PUSH operand** Adds a new item at top of stack
  - specify 16-bit source operand
  - operand may be register or memory
  - Stack grows "down" (to lower addresses):

    SP := SP – 2 **// adjust pointer**

    mem[SP] := operand **// copy value to top**

- **POP operand** Removes item from top of stack
  - specify 16-bit destination operand
  - operand may be register or memory

    operand := mem[SP] **// copy value to top**

    SP := SP + 2 **// adjust pointer**

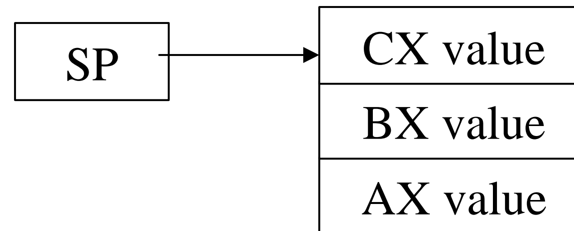
Intel Stack Instructions

- To read value in stack, index from the top (given by SS:SP)
  - Natural solution : `MOV  AX, [SP + constant]`
    - recall limitations on indirect addressing modes :
      - can only use BP, BX, SI, DI
      - **But … SS** default segment for indirect memory, access using **BP**
  - Needed solution
    - `MOV BP, SP`
    - `MOV AX, [BP+constant]`

Stack as a Storage Buffer

- The stack can be used to temporarily hold data
- **Example**: Suppose we need to save registers AX, BX and CX

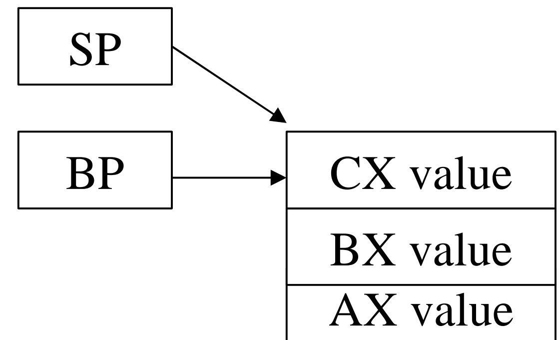
```
PUSH   AX
PUSH   BX
PUSH   CX
```



Stack as a Storage Buffer

- **Example**: Suppose we now need to access the saved value of AX.
  - We could POP off the values off until AX is reached.

  or
  - We could index into the stack.




```
MOV BP, SP

MOV  CX,[ BP + 4 ]  ;read saved AX
```