

The Stack

- 6 registers is limiting, so stack allows us to save and restore values
- like a stack of plates
- Last In First Out (LIFO)
- starts at high address and grows downwards
- putting too much data on the stack will cause it to overflow

The Stack Segment

- items are placed, or *pushed*, onto the stack
- items are removed, or *popped*, from the stack
- each program has its own stack
- SP (Stack Pointer) is the register for the stack
- **pushing** onto the stack causes the size of the item to be subtracted from the SP and all the bytes of the item are copied onto the stack
- **popping** from the stack causes the size of the item to be added to the SP (the bytes still remain on the stack but will be overwritten by the next push)

Why we have a stack?

- useful for storing context
 - one procedure can call another

Procedure called:

1. Push address of instruction after proc call onto stack
2. Parameters to the function are pushed onto the stack

Procedure finished:

1. Local variables will be popped off the stack
2. The parameters will be popped off the stack
3. *Return* pops the top value off the stack and puts into the IP

Why we have a stack?

Example:

Line	Code
10	<code>i = 3;</code>
11	<code>foo(i);</code>
12	<code>i = 4;</code>

- after line 10, the stack will have the address of the instructions composing line 12 pushed onto the stack
- then number 3 will be pushed onto the stack
- when foo is done, the number 3 is popped off the stack
- return puts the address of the first instruction for line 12 into IP

Introduction to Assembly Instructions

- many, but 20 common instructions
- generally:
 - 3 chars, an operand, a comma, an operand
 - a ; (semicolon) is used for commenting your code
- examples:
 - `mov ax, 10` ; put 10 into ax
 - `mov bx, 50` ; put 50 into bx

Push and Pop

PUSH

- puts a piece of data onto the stack

POP

- puts the piece of data from the top of the stack into a specified register or variable

Example:

push cx ; put cx on the stack

push ax ; put ax on the stack

pop cx ; put value from stack in cx

pop ax ; put value from stack in ax

Effect: exchanges the values in ax and cx

Note: *xchg ax, cx* would do the same

Types of Operand

- immediate
 - the number will be known at compilation
 - will always be the same e.g. '20'
- register
 - any general purpose or index register
 - e.g ax
- memory
 - a variable stored in memory

Common instructions

MOV

- MOV destination, source
- moves a value from one place to another
- e.g. `mov ax, bx`

INT

- INT interrupt_number
- calls system functions
- e.g. `int 21h ; call DOS service`
`int 10h ; call the video BIOS interrupt`

Common instructions

Declaring data

- Number1 db ? (unitialised byte)
- Number2 dw ? (unitialised word)
- Number3 db 1
- Number4 dw 2
- you cannot declare a variable as a byte/word and move the value into a 16-bit/8-bit register
- mov al, Number1 (ok) mov bl, Number2 (error)
- mov ax, Number1 (error) mov bx, Number2 (ok)

Common instructions

ADD

- ADD operand1, operand 2
- adds operand1 and operand2
- answer stored in operand1
- only operand2 can be immediate data (e.g. '20')

SUB

- SUB operand1, operand2
- subtracts operand2 from operand1, result in operand1
- only operand2 can be immediate data

Common instructions

MUL (unsigned)/IMUL (signed)

- MUL operand1
- multiplies AL or AX by operand1
- answer stored in AX (byte), DX:AX (word)
- EDX:EAX also available

DIV (unsigned)/IDIV (signed)

- DIV operand1
- divides AX (byte) DX:AX (word) by operand1
- AL answer, AH remainder (byte)
- AX answer, DX remainder (word)

Common instructions

INT

- most interrupts have more than one function
- must pass a number to the function via AH
- <http://spike.scu.edu.au/~barry/interrupts.html>

Message DB “Hello World!\$”

```
mov dx, OFFSET Message    ; DX contains the offset of message
mov ax, SEG Message       ; AX contains segment of message
mov ds, ax                ; DS:DX points to message
mov ah, 9                 ; function 9 - display string
int 21h                   ; call dos function
```

OFFSET/SEG tell the compiler that you want the offset/segment of the message put in the register, not the contents of the message