COAL — Lab Manual 01 & 02 (Complete solutions and step-by-step DEBUG sessions)

Objectives 1. Understand the DEBUG utility and how to use it. 2. Learn frequently used DEBUG commands. 3. Assemble, unassemble and trace simple assembly programs using DEBUG.

Instructor: Mr. Tariq Mehmood Butt (tariq.butt@pucit.edu.pk)

Quick orientation to DEBUG

- Launch DEBUG from DOS prompt: `I:\> debug` → prompt becomes `-`.
- Every DEBUG command is issued at the `-` prompt. You may use upper or lower case.
- To assemble instructions at a memory offset use `a address` (e.g. `a 0100`).
- To unassemble use `u address [range]`.
- To inspect memory use `d address1 address2` (Dump).
- To edit memory use `e address values...` (Enter) — hex bytes.
- To set or display registers use `r` or `r register` (e.g. `r es`).
- To run a program use `g`, to step use `t` (trace) or `p` (proceed — steps over CALLs).
- When in single-byte entry mode (no values on the `E` line) DEBUG prints the current byte in hex and waits for you to enter a new hex byte or press space to keep it.

  Note: In this manual every command that you type at the DEBUG prompt is shown on its own line (without the `I:\>` prompt). When I show an `a 0100` block, type those lines into DEBUG and finish the assembly input with a blank line.

Short DEBUG command cheat-sheet (most used)

- `?` — help screen
- `a address` — assemble starting at `address` (hex)
- `u [address] [range|Llength]` — unassemble
- `d [address1] [address2]` — dump memory (hex & ASCII)
- `e address data...` — enter bytes at `address` (hex values)
- `r [reg]` — display / change registers (AX, BX, CX, DX, SI, DI, BP, SP, IP, CS, DS, ES, FLAGS)
- `f range value` — fill memory block
- `m range address` — move memory block
- `s range value` — search memory
- `i port` — input from port
- `o port value` — output to port
- `t` — trace (single step). Shows registers after each step.
- `p` — proceed (step but step over CALL/INT)
- `g` — go (run from current IP)

- `q` — quit DEBUG

Lab Tasks — step-by-step solutions (use only DEBUG commands)

> **Important**: Before running a sequence, save any work. In DEBUG `a 0100` will assemble instructions starting at offset `0100` of the current code segment (CS). Most examples below use segment `4000` only where we explicitly set segment registers. When I show `4000:0100` that means segment 4000 and offset 0100.

Task 1 — Set ES = 0EE

Goal: *Write a sequence of commands to change the current value of ES register to* `0EE`.

Commands (at DOS prompt type `debug`, then at `-` prompt):

```
- r es
ES=0000  ; DEBUG shows current ES and waits for new value
00EE     ; type new value and press Enter
- r      ; verify registers (ES should now be 00EE)
```

Explanation: `r es` displays the current ES value and prompts you to type a new value. Enter `00EE` (hex) and press Enter. Then `r` shows the register list so you can confirm ES changed.

Task 2 — Display memory from 0100 for 80h bytes

Goal: Display the data in memory starting at offset `0100` for `80h` (128) bytes.

Command:

```
- d 0100 017F
```

Explanation: 128 bytes starting at 0x0100 end at 0x0100 + 0x80 - 1 = 0x017F. `D` displays hex bytes and ASCII on each line.

Task 3 — (a) Enter string `Hello` starting at offset 2 (b) Display only that message

(a) Enter `Hello` at offset 0002

You can do this in two ways: enter ASCII bytes (hex) or use single-byte entry mode.

Method A — Hex bytes (recommended predictable):

```
- e 0002 48 65 6C 6C 6F
```

Explanation: ASCII hex codes: `H`=48, `e`=65, `l`=6C, `l`=6C, `o`=6F.

Method B — Interactive entry mode:

```
- e 0002
0002: 00  ; DEBUG shows existing byte at 0002 and waits for your input
48 <space>   ; type 48 then press SPACE (or type 48 and press Enter to accept).
Continue for next bytes
65 <space>
6C <space>
6C <space>
6F <enter>
```

(b) Display just the message `Hello` you entered

Assuming you put it at offset `0002` and it's 5 bytes long:

```
- d 0002 0006
```

`D` will display hex bytes followed by ASCII on the right — you will see `48 65 6C 6C 6F   Hello`.

Task 4 — What does `-U CS:100 1E0` do?

Answer: `U CS:100 1E0` unassembles (disassembles) the bytes starting at `CS:0100` and decodes `1E0` (hex) bytes into 8086 mnemonics. `1E0h = 480` decimal — DEBUG will print the decoded instructions for that block (several screens worth). Use `u cs:100 1e0` when you want to view the machine code instructions in readable assembly form.

Task 5 — Which register refers to code?

Short answer: `CS` (Code Segment).
(Execution address = CS:IP)

Task 6 — Which command exits DEBUG?

Answer: `q` (Quit).

Task 7 — Run the given small code fragments in DEBUG and record flags

How to run (common steps): 1. `debug` 2. `a 0100` — start assembling at offset 0100 3. Type the three assembly lines for the subtask, then press an empty line to finish assembly 4. Use `r` to view registers or `t` to trace step by step and `r` after the final instruction to view flags.

Notes on flags we will report: CF (carry), ZF (zero), SF (sign), OF (overflow). (PF and AF can also be checked, but instructors usually ask the primary four.)

(i)

Code:

```
- a 0100
mov ax,FF12
mov bx,0012
add ax,bx

- t    ; step through until after ADD
- r    ; read registers and flags
```

Calculation & result: - ax initially `FF12h` (signed -238); add `0012h` (18) ⇒ result `FF24h`. - **CF = 0** (no carry out of 16 bits) - **ZF = 0** (result not zero) - **SF = 1** (MSB = 1 → negative in signed view) - **OF = 0** (no signed overflow: adding opposite signs not producing overflow)

(ii)

Code:

```
- a 0100
mov al,0001
```

```
dec al

- t
- r
```

Result: `AL` becomes `00` (zero). - **ZF = 1** (result zero) - **SF = 0** (sign = 0) - **OF = 0** (no signed overflow) - **CF is not affected by DEC** (it remains whatever it was before)

(iii)

Code:

```
- a 0100
mov al,FF
inc al

- t
- r
```

Result: `AL` goes from `FFh` to `00h`. - **ZF = 1** - **SF = 0** - **OF = 0** - **CF unchanged by INC**

(iv)

Code:

```
- a 0100
mov ax,0040
mov bx,0050
sub ax,bx

- t
- r
```

Result: `0040h - 0050h = FFF0h` (16-bit wraparound negative value) - **CF = 1** (borrow occurred) - **ZF = 0** - **SF = 1** (MSB = 1) - **OF = 0** (no signed overflow in this subtraction)

Task 8 — Assembly sequence (AX→1234, +1, copy to DX, subtract 1233 from DX, BH = DL, set AL=9)

Assembly:

```
- a 0100
mov ax,1234
add ax,1
mov dx,ax
sub dx,1233
mov bh,dl
mov al,09

; blank line to finish
```

What to do: - After assembling, use `t` to step and `r` to inspect AX, DX, BX(BH), AL etc. - Expected values after run: - AX = 1235 - DX = AX - 1233 = 0002 - BH = DL (DL is low byte of DX => 02), so BH = 02 - AL = 09

Task 9 — Assembly sequence (AX=4000h; add AX to AX; subtract 0FFFFh; inc AX; dec AX)

Assembly:

```
- a 0100
mov ax,4000
add ax,ax          ; AX = 8000
sub ax,FFFF        ; AX = 8000 - FFFF = 8001 (since -FFFF is +1)
inc ax             ; AX = 8002
dec ax             ; AX = 8001

; blank line
```

Explanation: Subtracting `FFFFh` is equivalent to adding `0001h` (mod 65536). Track registers with `t` and verify with `r`.

Task 10 — Exchange AX and BX

Simplest assembly:

```
- a 0100
mov ax,1111
mov bx,2222
xchg ax,bx

; blank line
```

Explanation: After `xchg ax,bx`, AX will contain the original BX value and BX will contain the original AX value. If `xchg` is not allowed in some micro-modes you can swap via `xchg ax,bx` or using a temporary register like `push` / `pop`.

Alternate (push/pop) method:

```
push ax
mov ax,bx
pop bx
```

Task 11 — Copy an 8-byte array from 4000:0100..0107 → 4000:0200..0207

Plan: initialize source with `E`, assemble a small copy using `rep movsb`, run, then `D` to verify.

Step A — Initialize the source memory (example values):

```
- e 4000:0100 11 22 33 44 55 66 77 88
- d 4000:0100 4000:0107   ; verify source
```

Step B — Assembly (copy routine using DS & ES both = 4000):

```
- a 0100
mov ax,4000
mov ds,ax
mov es,ax
mov si,0100
mov di,0200
mov cx,08
cld
rep movsb
ret

; blank line
```

Step C — Run and verify:

```
- t    ; step until the routine runs OR use 'g' to run until RET
- d 4000:0200 4000:0207  ; verify destination contains 11 22 33 44 55 66 77 88
```

Task 12 — Copy 8-byte source (0100..0107) in *reverse order* into 0200..0207

Goal: target[0200] = source[0107], target[0201] = source[0106], … , target[0207] = source[0100].

Method (loop):

```
- a 0100
mov ax,4000
mov ds,ax
mov es,ax
mov si,0107     ; start at last source byte
mov di,0200     ; place into first destination byte
mov cx,08
cld
L1:
mov al,[si]
mov [di],al
dec si
inc di
loop L1
ret

; blank line
```

Run & verify:

```
- t  ; step through, or use g to run the whole routine
- d 4000:0200 4000:0207  ; verify reversed copy
```

Task 13 — Swap (element-wise) contents of two 8-byte arrays at 0100..0107 and 0200..0207

Plan: For i = 0..7 swap the bytes in place using `AL` and `BL` as temporaries.

Initialize arrays (example):

```
- e 4000:0100 01 02 03 04 05 06 07 08
- e 4000:0200 AA BB CC DD EE FF 00 11
```

Assembly (swap loop):

```
- a 0100
mov ax,4000
mov ds,ax
mov es,ax
mov si,0100
mov di,0200
mov cx,08
cld
L1:
mov al,[si]
mov bl,[di]
mov [si],bl
mov [di],al
inc si
inc di
loop L1
ret

; blank line
```

Run & verify with `d` on both arrays.

Task 14 — Reverse-SWAP two 8-byte arrays

Interpretation used here: exchange A[i] with B[7-i] (i.e. reverse index on the second array while swapping).

Assembly:

```
- a 0100
mov ax,4000
mov ds,ax
mov es,ax
mov si,0100    ; pointer into A (increasing)
mov di,0207    ; pointer into B (decreasing)
mov cx,08
cld
L1:
mov al,[si]
mov bl,[di]
mov [si],bl
mov [di],al
inc si
```

```
dec di
loop L1
ret

; blank line
```

Verify: `d 4000:0100 4000:0107` and `d 4000:0200 4000:0207`.

How to step/trace and inspect flags & memory

- Use `t` to execute a single instruction and then `r` to view registers and flags.
- Use `p` to step over subroutine calls (`call`) so you do not step into called routine.
- Use `d seg:offset range` to watch memory bytes change after each `t` step.

Example: after assembling code at `0100`, do:

```
- r ip
- t   ; executes first instruction
- r   ; inspect registers/flags
- d 4000:0100 4000:0107   ; inspect memory after steps
```

Common pitfalls & tips

- Always use hex values. If you write `mov ax,1234` DEBUG treats `1234` as hex.
- When editing memory (`E`) you must enter hex bytes. ASCII to hex: use a small table or calculator (H=48, e=65 etc.).
- Segment registers can only be loaded from a general register (e.g., `mov ax,4000` then `mov ds,ax`).
- `rep movsb` uses DS:SI → ES:DI and CX count; be sure DS, ES values are correct.
- If a loop does not terminate, press `Ctrl+C` to break out (in real DOS) and inspect your code.
- When using `loop` label the loop (DEBUG supports labels) or use relative jumps; if in doubt use `dec cx` + `jnz` approach.

Short checklist for each task when you hand in the lab

1. Show the sequence of DEBUG commands you typed.
2. Show the `D` dumps (before and after) for memory-based tasks to verify correctness.
3. For register/flag tasks, show `R` output after executing the instruction that changes flags.
4. For loops, use `T` to show an example trace step (before & after one iteration) and then `G` or complete `T` runs to finish.

If you want, I can also: - produce a printable PDF of this manual, or - walk you interactively through **any single task** (I will give the exact commands you must type and what you will see) — tell me which task to simulate.

Good luck — open `COAL Lab Manual Solutions - Lab 01 & 02` (this document) and tell me which task you'd like to run first and I will walk you step-by-step.