Assembly Language Concepts Guide

Program Structure

Origin Directive
- `ORG 100H`: Specifies the starting address of the program (usually 100H for COM files)
- Example: `org 100h`

Segments
- `.DATA`: Data segment for storing variables
- `.CODE`: Code segment for program instructions
- Example:
```asm
.DATA
  message db 'Hello$'
.CODE
  MAIN PROC
    ; code goes here
  MAIN ENDP
```

Procedures
- `PROC`: Defines the start of a procedure
- `ENDP`: Marks the end of a procedure
- Example:
```asm
FINDAVERAGE PROC
  ; procedure code
  RET  ; return to caller
FINDAVERAGE ENDP
```

Program Termination
- `END MAIN`: Marks the end of the program
- `RET`: Return from procedure or program

Data Definitions

Types
- `DB` (Define Byte): Reserves 1 byte of storage (8-bits)
  - Example: `message DB 'Hello$'`
  - Example: `count DB 5`

- `DW` (Define Word): Reserves 2 bytes of storage (16-bits)
  - Example: `A DW 1,2,3,4,5`
  - Example: `n DW ?` (uninitialized variable)

Arrays and Strings
- Arrays defined with multiple values: `A DW 1,2,3,4,5`
- String termination with '$': `message DB 'Hello$'`
- Duplicate values: `array DB 6 DUP(?)` (6 uninitialized bytes)
- New line characters: `0Dh, 0Ah` (carriage return and line feed)

Registers and Their Uses

General Purpose Registers
- `AX`: Accumulator register (arithmetic, I/O operations)
- `BX`: Base register (often used for addressing)
- `CX`: Counter register (loop counting)
- `DX`: Data register (I/O, multiplication/division)

Segment Registers
- `DS`: Data segment register
  - Example: `MOV AX, @DATA` then `MOV DS, AX` to set up the data segment

Register Parts
- `AH/AL`: High/Low bytes of AX
- `BH/BL`: High/Low bytes of BX
- `CH/CL`: High/Low bytes of CX
- `DH/DL`: High/Low bytes of DX

Memory Addressing

Offset Addressing
- `OFFSET`: Gets the address of a variable
  - Example: `MOV SI, OFFSET array`

Register Indirect Addressing
- Using registers to point to memory:
  - `[SI]`: Content at address in SI
  - Example: `MOV AL, [SI]`

Instructions

Data Movement
- `MOV`: Moves data between registers or memory
  - Example: `MOV AX, BX` (copy BX to AX)
  - Example: `MOV [SI], AL` (store AL at address in SI)

- `LEA` (Load Effective Address): Gets the address of a variable
  - Example: `LEA SI, array` (similar to MOV SI, OFFSET array)

- `XCHG`: Exchanges the contents of two registers or memory locations

Arithmetic Operations
- `ADD`: Addition
  - Example: `ADD AX, BX` (AX = AX + BX)

- `SUB`: Subtraction
  - Example: `SUB BX, 225` (BX = BX - 225)

- `MUL`: Unsigned multiplication
  - Example: `MUL AX` (DX:AX = AX * operand)

- `DIV`: Unsigned division
  - Example: `DIV CL` (AL = AX / CL, AH = remainder)
  - Divides AX by the operand, quotient in AL, remainder in AH

- `INC`: Increment by 1
  - Example: `INC SI` (SI = SI + 1)

- `DEC`: Decrement by 1
  - Example: `DEC CX` (CX = CX - 1)

Logical Operations
- `AND`: Bitwise AND

- `OR`: Bitwise OR
- `XOR`: Bitwise XOR
  - Example: `XOR AX, AX` (zeros out AX, faster than MOV AX, 0)

- `CMP`: Compare two values (sets flags based on comparison)
  - Example: `CMP AL, 'a'` (compare AL with character 'a')

- `TEST`: Bitwise AND without storing the result (sets flags)
  - Example: `TEST DX, DX` (check if DX is zero)

Control Flow

Unconditional Jump
- `JMP`: Jumps to a label
  - Example: `JMP ENDD` (transfer control to ENDD label)

Conditional Jumps
- `JZ`/`JE`: Jump if Zero/Equal
- `JNZ`/`JNE`: Jump if Not Zero/Not Equal
- `JL`: Jump if Less
- `JLE`: Jump if Less or Equal
- `JG`: Jump if Greater
- `JGE`: Jump if Greater or Equal
- `JB`: Jump if Below (unsigned)
- `JA`: Jump if Above (unsigned)
- Example:
```asm
CMP AL, 'a'
JGE LOWERCASE  ; Jump if AL >= 'a'
```

Loop Instructions
- `LOOP`: Decrement CX and jump if CX ≠ 0
  - Example:
```asm
MOV CX, 5
MYLABEL:
  ; code to repeat 5 times
  LOOP MYLABEL
```

```

## Input/Output Operations

### DOS Interrupt 21H
- `INT 21H`: Call DOS function specified in AH
- Common functions:
  - `AH=1`: Read character with echo (character in AL)
  - `AH=2`: Display character in DL
  - `AH=9`: Display string (pointed by DS:DX, terminated with '$')
- Example:
```asm
MOV AH, 1    ; read character
INT 21H      ; character stored in AL

MOV DL, AL   ; move character to DL
MOV AH, 2    ; display character
INT 21H
```

Stack Operations

- `PUSH`: Push a value onto stack
  - Example: `PUSH AX`

- `POP`: Pop a value from stack
  - Example: `POP BX`

Macros

- Reusable code blocks defined once and used multiple times
- Example:
```asm
MDSPLY_STRING MACRO STRING
  MOV DX, OFFSET STRING
  CALL DSPLY_STRING
ENDM
```
```

Common Techniques

String Processing
- Using SI to iterate through strings
- Checking for special characters (spaces, '$', etc.)
- Example:
```asm
LEA SI, input_string
ITERATE:
  CMP [SI], '$'
  JZ END_PROCESS
  ; process character
  INC SI
  JMP ITERATE
END_PROCESS:
```

Case Conversion
- Uppercase to lowercase: Add 32 (`ADD AL, 32`)
- Lowercase to uppercase: Subtract 32 (`SUB AL, 32`)
- ASCII conversion: Add/subtract '0' to convert between ASCII and numeric values

Array Operations
- Bubble sort (as seen in bubblesort.asm)
- Finding average and largest/smallest values

Common Mistakes to Avoid

1. **Register Preservation**: Not preserving registers in procedures (use PUSH/POP)
2. **Data Segments**: Forgetting to initialize DS register
3. **Division by Zero**: No automatic check for division by zero
4. **String Termination**: Forgetting '$' at the end of strings for DOS output
5. **Overflow/Underflow**: Not accounting for arithmetic overflow
6. **Return from Procedures**: Missing RET instruction
7. **Comparison Order**: Getting the order wrong in CMP operations
8. **Register Size**: Using the wrong register size (AX vs AL)
9. **Loop Counter**: Not initializing CX before LOOP instructions

```
## Tips and Best Practices

1. **Comments**: Add comments for clarity
2. **Modularity**: Use procedures for reusable code
3. **Register Usage**: Be consistent with register usage
4. **Clear Labels**: Use descriptive label names
5. **Data Organization**: Group related data together
6. **Error Handling**: Include error checking where possible
```