**COMP 3350 Exam 2 Study Guide**

**Table of Contents:**

* CHP 5: Pages 1 – 4
* CHP 6: Pages 5 – 10
* CHP 7: Pages 10 – 14
* CHP 8: Pages 14 - 19

**CHP 5 - Procedures:**

* Stack Operations:
  + Runtime Stack: Every thing is added & removed from the top (LIFO Structure).
    - Managed by the CPU, using two registers – SS (Stack Segment) & ESP (Stack Pointer, SP in Real-Address Mode).
  + PUSH Operation: A 32-bit push operation decrements the stack pointer by 4 & copies a value into the location pointed to by the stack pointer.
    - Syntax: PUSH r/m16, PUSH r/m32, PUSH imm32
  + POP Operation: Copies value at stack[ESP] into a register or variable.
    - Adds n to ESP, where n is either 2 or 4. The value of n depends on the attribute of the operand receiving the data.
    - Syntax: POP r/m16, POP r/m32
  + Using PUSH & POP: Save & restore registers when they contain important values. PUSH & POP instructions occur in the opposite order.
    - Ex: push esi ; push registers

Push ecx

Push ebx

Mov esi, OFFSET dwordVal ; display some memory

Mov ecx, LENGTHOF dwordVal

Mov ebx, TYPE dwordVal

Call DumpMem

Pop ebx ; restore registers

Pop ecx

Pop esi

* + Nested Loops: When creating a nested loop, push the outer loop counter before entering the inner loop (remember to pop it after the second loop completes).
  + Reversing a String:
    - Use a loop with indexed addressing.
    - Push each character on the stack.
    - Start at the beginning of the string, pop the stack in reverse order, insert each character back into the string.
  + Related Instructions:
    - PUSHFD & POPFD: Push & pop the EFLAGS register
    - PUSHAD: Pushes the 32-bit general-purpose registers on the stack; Order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.
    - POPAD: Pops the same registers off the stack in reverse order; PUSHA & POPA do the same for 16-bit registers.
* Defining & Using Procedures:
  + Creating Procedures:
    - Large problems can be divided into smaller tasks to make them more manageable.
    - A procedure is the ASM equivalent of a Java or C++ function.
    - Following is an assembly language procedure named sample:
      * Sample PROC

.

Ret

Sample ENDP

* + - Documenting Procedures:
      * A description of all tasks accomplished by the procedure.
      * Receives: A list of input parameters; state their usage & requirements.
      * Returns: A description of values returned by the procedure.
      * Requires: Optional list of requirements called preconditions that must be satisfied before the procedure is called.
  + CALL & RET Instructions:
    - The CALL instruction calls a procedure.
      * Pushes offset of next instruction on the stack.
      * Copies the address of the called procedure into EIP.
    - The RET instruction returns from a procedure.
      * Pops top of stack into EIP.
  + Local & Global Labels: A local label is visible only to statements inside the same procedure. A global label is visible everywhere.
  + Procedure Parameters:
    - A good procedure might be usable in many different programs, but not if it refers to specific variable names.
    - Parameters help to make procedure flexible because parameter values can change at runtime.
  + USES Operator: Lists the registers that will be preserved.
    - Syntax: ArraySum PROC USES esi ecx
      * Mov eax,0 ; set the sum to zero

Etc.

* + - Replaces the push & pop operations.
* Linking to an External Library:
  + What is a Link Library:
    - A file containing procedures that have been compiled into machine code; constructed from one or more OBJ files.
    - To build a library…
      * Start with one or more ASM source files.
      * Assemble each into an OBJ file.
      * Create an empty library file (extension: .LIB).
      * Add the OBJ file(s) to the library file, using the Microsoft LIB utility.
  + How the Linker Works:
    - Your programs link to Irvine32.lib using the linker command inside a batch file named make32.bat.
    - Notice the two LIB files: Irvine32.lib, & kernel32.lib.
      * The latter is part of the Microsoft Win32 Software Development Kit (SDK).
* The Irvine32 Library:
  + Calling Irvine32 Library Procedures:
    - Call each procedure using the CALL instruction. Some procedures require input arguments. The INCLUDE directive copies in the procedure prototypes (declarations).
  + Library Procedures – Overview:
    - CloseFile: Closes an open disk file.
    - Clrscr: Clears console, locates cursor at upper left corner.
    - CreateOutputFile: Creates new disk file for writing in output mode.
    - Crlf: Writes end of line sequence to standard output.
    - Delay: Pauses program execution for n millisecond interval.
    - DumpMem: Writes block of memory to standard output in hex.
    - DumpRegs: Displays general-purpose registers & flags (hex).
    - GetCommandTail: Copies command-line args into array of bytes.
    - GetDateTime: Gets the current date & time from the system.
    - GetMaxXY: Gets number of cols, rows in console window buffer.
    - GetMSeconds: Returns milliseconds elapsed since midnight.
    - GetTextColor: Returns active foreground & background text colors in the console window.
    - Gotoxy: Locates cursor at row & column on the console.
    - IsDigit: Sets Zero flag if AL contains ASCII code for decimal digit (0 – 9).
    - MsgBox, MsgBoxAsk: Display popup message boxes.
    - OpenInputFiles: Opens existing file for input.
    - ParseDecimal32: Converts unsigned integer string to binary.
    - ParseInteger32: Converts signed integer string to binary.
    - Random32: Generates 32-bit pseudorandom integer in the range 0 to FFFFFFFFh.
    - Randomize: Seeds the random number generator.
    - RandomRange: Generates a pseudorandom integer within a specified range.
    - ReadChar: Reads a single character from standard input.
    - ReadDec: Reads 32-bit unsigned decimal integer from keyboard.
    - ReadFromFile: Reads input disk file into buffer.
    - ReadHex: Reads 32-bit hexadecimal integer from keyboard.
    - ReadInt: Reads 32-bit signed decimal integer from keyboard.
    - ReadKey: Reads character from keyboard input buffer.
    - ReadString: Reads string from stdin, terminated by [Enter].
    - SetTextColor: Sets foreground/background colors of all subsequent text output to the console.
    - Str\_Compare: Compares two strings.
    - Str\_Copy: Copies a source string to a destination string.
    - Str\_Length: Returns the length of a string in EAX.
    - Str\_Trim: Removes unwanted characters from a string.
    - Str\_Ucase: Converts a string to uppercase letters.
    - WaitMsg: Displays message, waits for Enter key to be pressed.
    - WriteBin: Writes unsigned 32-bit integer in ASCII binary format.
    - WriteBinB: Writes binary integer in byte, word, or doubleword format.
    - WriteChar: Writes a single character to standard output.
    - WriteDec: Writes unsigned 32-bit integer in decimal format.
    - WriteHex: Writes an unsigned 32-bit integer in hexadecimal format.
    - WriteHexB: Writes byte, word, or doubleword in hexadecimal format.
    - WriteInt: Writes signed 32-bit integer in decimal format.
    - WriteStackFrame: Writes the current procedure’s stack frame to the console.
    - WriteStackFrameName: Writes the current procedure’s name & stack frame to the console.
    - WriteString: Writes null-terminated string to console window.
    - WriteToFile: Writes buffer to output file.
    - WriteWindowsMsg: Displays most recent error message generated by MS-Windows.
* Summary:
  + Procedure: Named block of executable code.
  + Runtime Stack: LIFO structure.
    - Holds return addresses, parameters, local variables.
    - PUSH: Add value to stack.
    - POP: Remove value from stack.
  + Use the Irvine32 library for all standard I/O & data conversion.

**CHP 6 – Conditional Processing:**

* Boolean & Comparison Instructions:
  + Status Flags – Review:
    - Zero Flag: Set when the result of an operation equals zero.
    - Carry Flag: Set when an instruction generates a result that is too large (or too small) for the destination operand.
    - Sign Flag: Set if the destination operand is negative, & it is clear if the destination operand is positive.
    - Overflow Flag: Set when an instruction generates an invalid signed result (bit 7 carry is XORed with bit 6 Carry).
    - Parity Flag: Set when an instruction generates an even number of 1 bits in the low byte of the destination operand.
    - Auxiliary Carry: Set when an operation produces a carry out from bit 3 to bit 4.
  + AND Instruction: Performs a Boolean AND operation between each pair of matching bits in two operands.
    - Syntax: AND destination, source
  + OR Instruction: Performs a Boolean OR operation between each pair of matching bits in two operands.
    - Syntax: OR destination, source
  + XOR Instruction: Performs a Boolean exclusive-OR operation between each pair of matching bits in two operands.
    - Syntax: XOR destination, source
  + NOT Instruction: Performs a Boolean NOT operation on a single destination operand.
    - Syntax: NOT destination
  + Bit-Mapped Sets:
    - Binary bits indicate set membership.
    - Efficient use of storage.
    - Also known as bit vectors.
    - Bit-Mapped Set Operations:
      * Set Complement:
        + Mov eax, setX
        + Not eax
      * Set Intersection:
        + Mov eax, setX
        + And eax, setY
      * Set Union:
        + Mov eax, setX
        + Or eax, setY
  + Applications:
    - Task: Convert the character in AL to upper case.
    - Solution: Use the AND instruction to clear bit 5.
    - Task: Convert a binary decimal byte into its equivalent ASCII decimal digit.
    - Solution: Use the OR instruction to set bits 4 & 5.
    - Task: Turn on the keyboard CapsLock key.
    - Solution: Use the OR instruction to set bit 6 in the keyboard flag byte at 0040:0017h in the BIOS data area.
    - Task: Jump to a label if an integer is even.
    - Solution: AND the lowest bit with a 1. If the result is Zero, the number was even.
    - Task: Jump to a label if the value in AL is not zero.
    - Solution: OR the byte with itself, then use the JNZ (jump if not zero) instruction.
  + TEST Instruction:
    - Performs a nondestructive AND operation between each pair of matching bits in two operands.
    - No operands are modified, but the Zero flag is affected.
    - Ex: Jump to a label if either bit 0 or bit 1 in AL is set.
    - test al, 00000011b
    - Jnz ValueFound
    - Ex: Jump to a label of neither bit 0 nor bit 1 in AL is set.
    - Test al, 00000011b
    - Jz ValueNotFound
  + CMP Instruction:
    - Compares the destination operand to the source operand.
      * Nondestructive subtraction of source from destination (destination operand is not changed).
    - Syntax: CMP destination, source
    - Ex: Destination == source
    - Mov al, 5
    - Cmp al, 5 ; Zero flag set
    - Ex: destination < source
    - Mov al, 4
    - Cmp al, 5 ; Carry flag set
    - Ex: Destination > source
    - Mov al, 6
    - Cmp al, 5 ; ZF = 0, CF = 0
    - The next two comparisons shown are performed with signed integers.
    - Ex: Destination > source
    - Mov al, 5
    - Cmp al, -2 ; Sign flag == Overflow flag
    - Ex: Destination < source
    - Mov al, -1
    - Cmp al, 5 ; Sign flag != Overflow flag
  + Boolean Instructions in 64-Bit Mode:
    - 64-bit boolean instructions, for the most part, work the same as 32-bit instructions.
    - If the source operand is a constant whose size is less than 32 bits & the destination is the lower part of a 64-bit register or memory operand, all bits in the destination operand are affected.
    - When the source is a 32-bit constant or register, only the lower 32 bits of the destination operand are affected.
* Conditional Jumps:
  + Jcond Instruction: A conditional jump instruction branches to a label when specific register or flag conditions are met.
    - Specific Jumps:
      * JB, JC: Jump to a label if the Carry flag is set.
      * JE, JZ: Jump to a label if the Zero flag is set.
      * JS: Jump to a label if the Sign flag is set.
      * JNE, JNZ: Jump to a label if the Zero flag is clear.
      * JECXZ: Jump to a label if ECX = 0.
  + Jcond Ranges:
    - Prior to the 386: Jump must be within -128 to +127 bytes from current location counter.
    - X86 Processors: 32-bit offset permits jump anywhere in memory.
  + Jumps Based on Specific Flags:
    - JZ: Jump if zero; ZF = 1
    - JNZ: Jump if not zero; ZF = 0
    - JC: Jump if carry; CF = 1
    - JNC: Jump if not carry; CF = 0
    - JO: Jump if overflow; OF = 1
    - JNO: Jump if not overflow; OF = 0
    - JS: Jump if signed; SF = 1
    - JNS: Jump if not signed; SF = 0
    - JP: Jump if parity (even); PF = 1
    - JNP: Jump if not parity (odd); PF = 0
  + Jumps Based on Equality:
    - JE: Jump if equal (leftOp = rightOp)
    - JNE: Jump if not equal (leftOp != rightOp)
    - JCXZ: Jump if CX = 0
    - JECXZ: Jump if ECX = 0
  + Jumps Based on Unsigned Comparisons:
    - JA: Jump if above (if leftOp > rightOp)
    - JNBE: Jump if not below or equal (same as JA)
    - JAE: Jump if above or equal (if leftOp >= rightOp)
    - JNB: Jump if not below (same as JAE)
    - JB: Jump if below (if leftOp < rightOp)
    - JNAE: Jump if not above or equal (same as JB)
    - JBE: Jump if below or equal (if leftOp <= rightOp)
    - JNA: Jump if not above (same as JBE)
  + Jumps Based on Signed Comparisons:
    - JG: Jump if greater (if leftOp > rightOp)
    - JNLE: Jump if not less than or equal (same as JG)
    - JGE: Jump if greater than or equal (if leftOp >= rightOp)
    - JNL: Jump if not less (same as JGE)
    - JL: Jump if less (if leftOp < rightOp)
    - JNGE: Jump if not greater than or equal (same as JL)
    - JLE: Jump if less than or equal (if leftOp <= rightOp)
    - JNG: Jump if not greater (same as JLE)
  + Applications:
    - Task: Jump to a label if unsigned EAX is greater than EBX.
    - Solution: Use CMP, followed by JA.
    - Task: Jump to a label if signed EAX is greater than EBX
    - Solution: Use CMP, followed by JG
    - Task: Jump to label L1 if bits 0, 1, & 3 in AL are all set.
    - Solution: Clear all bits except bits 0, 1, & 3. Then compare the result with 00001011 binary.
  + BT (Bit Test) Instruction:
    - Copies bit n from an operand into the Carry flag.
    - Syntax: BT bitBase, n
      * bitBase may be r/m16 or r/m32
      * n may be r16, r32, or imm8
* Conditional Loop Instructions:
  + LOOPZ & LOOPE:
    - Syntax: LOOPE destination & LOOPZ destination
    - Logic:
      * ECX 🡨 ECX – 1
      * If ECX > 0 & ZF = 1, jump to destination
    - Useful when scanning an array for the first element that does not match a given value.
  + LOOPNZ & LOOPNE:
    - LOOPNZ (LOOPNE) is a conditional loop instruction.
    - Syntax: LOOPNZ destination & LOOPNE destination
    - Logic:
      * ECX 🡨 ECX – 1
      * If ECX > 0 & ZF = 0, jump to destination
    - Useful when scanning an array for the first element that matches a given value.
* Conditional Structures:
  + Block-Structured IF Statements: Assembly language programmers can easily translate logical statements written in C++/Java into assembly language.
  + Compound Expression with AND: When implementing the logical AND operator, consider that HLLs use short-circuit evaluation.
  + Compound Expression with OR: When implementing the logical OR operator, consider that HLLs use short-circuit evaluation.
  + WHILE Loops: A WHILE loop is really an IF statement followed by the body of the loop, followed by an unconditional jump to the top of the loop.
  + Table-Driven Selection:
    - Table-driven selection uses a table lookup to replace a multiway selection structure.
    - Create a table containing lookup values & the offsets of labels or procedures.
    - Use a loop to search the table.
    - Suited to a large number of comparisons.
    - Step 1: Create a table containing lookup values & procedure offsets.
    - Step 2: Use a loop to search the table. When a match is found, call the procedure offset stored in the current table entry.
* Application – Finite-State Machines:
  + A finite-state machine (FSM) is a graph structure that changes state based on some input. Also called a state-transition diagram.
  + We use a graph to represent an FSM, with squares or circles called nodes, & lines with arrows between the circles called edges.
  + A FSM is a specific instance of a more general structure called a directed graph.
  + Three basic states, represented by nodes:
    - Start state.
    - Terminal state(s).
    - Nonterminal state(s).
  + Finite-State Machine:
    - Accepts any sequence of symbols that puts it into an accepting (final) state.
    - Can be used to recognize, or validate a sequence of characters that is governed by language rules (called a regular expression).
    - Advantages:
      * Provides visual tracking of program’s flow of control.
      * Easy to modify.
      * Easily implemented in assembly language.
  + IsDigit Procedure: Receives a character in AL. Sets the Zero flag if the character is a decimal digit.
* Conditional Control Flow Directives:
  + Creating IF Statements:
    - Runtime Expressions:
      * .IF, .ELSE, .ELSEIF, & .ENDIF can be used to evaluate runtime expressions & create block-structured IF statements.
      * MASM generates “hidden” code for you, consisting of code labels, CMP & conditional jump instructions.
    - .REPEAT Directive: Executes the loop body before testing the loop condition associated with the .UNTIL directive.
    - .WHILE Directive: Tests the loop condition before executing the loop body. The .ENDW directive marks the end of the loop.
* Summary:
  + Bitwise instructions (AND, OR, XOR, NOT, TEST); manipulate individual bits in operands.
  + CMP: Compares operands using implied subtraction; sets condition flags.
  + Conditional Jumps & Loops:
    - Equality: JE, JNE.
    - Flag Values: JC, JZ, JNC, JP, …
    - Signed: JG, JL, JNG, …
    - Unsigned: JA, JB, JNA, …
    - LOOPZ, LOOPNZ, LOOPE, LOOPNE.
  + Flowcharts: Logic diagramming tool.
  + Finite-State Machine: Tracks state changes at runtime.

**CHP 7 – Integer Arithmetic:**

* Shift & Rotate Instructions:
  + Logical Shift: Shifts every bit right and fills the newly created position with zero.
  + Arithmetic Shift: Shifts every bit right and fills the newly created bit position with a copy of the number’s sign bit.
  + SHL Instruction: Performs a logical left shift on the destination operand, filling the lowest bit with 0.
    - Operand Types for SHL: SHL reg, imm8, SHL mem, imm8, SHL reg, CL, SHL mem, CL
    - Fast Multiplication: Shifting left 1 bit multiplies a number by 2.
      * Shifting left n bits multiplies the operand by 2n.
      * For example, 5 \* 22 = 20
  + SHR Instruction: Performs a logical right shift on the destination operand. The highest bit position is filled with a zero.
    - Shifting right n bits divides the operand by 2n.
  + SAL & SAR Instructions:
    - SAL (Shift Arithmetic Left): Identical to SHL.
    - SAR (Shift Arithmetic Right): Performs a right arithmetic shift on the destination operand.
    - An arithmetic shift preserves the number’s sign.
  + ROL (Rotate) Instruction: Shifts each bit to the left.
    - The highest bit is copied into both the Carry flag & into the lowest bit.
    - No bits are lost.
  + ROR (Rotate Right) Instruction: Shifts each bit to the right.
    - The lowest bit is copied into both the Carry flag & into the highest bit.
    - No bits are lost.
  + RCL (Rotate Carry Left): Shifts each bit to the left.
    - Copies the Carry flag to the least significant bit.
    - Copies the most significant bit to the Carry flag.
  + RCR (Rotate Carry Right): Shifts each bit to the right.
    - Copies the Carry flag to the most significant bit.
    - Copies the least significant bit to the Carry flag.
  + SHLD Instruction: Shifts a destination operand a given number of bits to the left.
    - The bit positions opened up by the shift are filled by the most significant bits of the source operand.
    - The source operand is not affected.
    - Syntax: SHLD destination, source, count
    - Operand Types: SHLD reg16/32, reg16/32, imm8/CL, SHLD mem16/32, reg16/32, imm8/CL
  + SHRD Instruction: Shifts a destination operand a given number of bits to the right.
    - The bit positions opened up by the shift are filled by the least significant bits of the source operand.
    - The source operand is not affected.
    - Syntax: SHRD destination, source, count
    - Operand Types: SHD reg16/32, reg16/32, imm8/CL, SHLD mem16/32, reg16/32, imm8/CL
* Shift & Rotate Applications:
  + Shifting Multiple Doublewords: Programs sometimes need to shift all bits within an array, as one might when moving a bitmapped graphic image from one screen location to another.
  + Binary Multiplication:
    - We already know that SHL performs unsigned multiplication efficiently when the multiplier is a power of 2.
    - You can factor any binary number into powers of 2.
  + Displaying Binary Bits:
    - Algorithm: Shift MSB into the Carry flag; if CF = 1, append a “1” character to a string; otherwise, append a “0” character. Repeat in a loop, 32 times.
* Multiplication & Division Instructions:
  + MUL (Unsigned Integer Multiplication) Instruction: In 32-bit mode, MUL (unsigned multiply) instruction multiplies an 8-, 16-, or 32-bit operand by either AL, AX, or EAX.
    - Instruction Formats: MUL r/m8, MUL r/m16, MUL r/m32.
  + IMUL (Signed Integer Multiplication) Instruction: Multiplies an 8-, 16-, or 32-bit signed operand by either AL, AX, or EAX.
    - Preserves the sign of the product by sign-extending it into the upper half of the destination register.
  + DIV (Unsigned Integer Division) Instruction: Performs 8-, 16-, or 32-bit division on unsigned integers.
    - A single operand is supplied (register or memory operand), which is assumed to be the divisor.
    - Instruction Formats: DIV reg/mem8, DIV reg/mem16, DIV reg/mem32.
  + IDIV (Signed Integer Division) Instruction: Performs signed integer division; same syntax & operands as DIV instruction.
    - Signed integers must be sign-extended before division takes place.
    - Fill high byte/word/doubleword with a copy of the low byte/word/doubleword’s sign bit.
    - For example, the high byte contains a copy of the sign bit from the low byte.
  + CBW, CWD, CDQ Instructions: Provide important sign-extension operations.
    - CBW (Convert Byte to Word): Extends AL into AH.
    - CWD (Convert Word to Doubleword): Extends AX into DX.
    - CDQ (Convert Doubleword to Quadword): Extends EAX into EDX.
* Extended Addition & Subtraction:
  + Extended Precision Addition:
    - Adding two operands that are longer than the computer’s word size (32 bits); virtually no limit to the size of the operands.
    - The arithmetic must be performed in steps; the Carry value from each step is passed on to the next step.
  + ADC (Add with Carry) Instruction: Adds both a source operand and the contents of the Carry flag to a destination operand.
    - Operands are binary values; same syntax as ADD, SUB, etc.
  + SBB (Subtract with Borrow) Instruction: Subtracts both a source operand & the value of the Carry flag from a destination operand.
    - Syntax: Same as for the ADC instruction.
* ASCII & UnPacked Decimal Arithmetic:
  + Binary-Coded Decimal:
    - Binary-coded decimal (BCD) integers use 4 binary bits to represent each decimal digit.
    - A number using unpacked BCD representation stores a decimal digit in the lower four bits of each byte.
  + ASCII Decimal: A number using ASCII Decimal representation stores a single ASCII digit in each byte.
  + AAA (ASCII Adjust after Addition) Instruction: Adjusts the binary result of an ADD or ADC instruction. It makes the result in AL consistent with ASCII decimal representation.
    - The Carry value, if any, ends up in AH.
  + AAS (ASCII Adjust after Subtraction) Instruction: Adjusts the binary result of any SUB or SBB instruction. It makes the result in AL consistent with ASCII decimal representation.
    - It places the Carry value, if any, in AH.
  + AAM (ASCII Adjust after Multiplication) Instruction: Adjusts the binary result of a MUL instruction. The multiplication must have been performed on unpacked BCD numbers.
  + AAD (ASCII Adjust before Division) Instruction: Adjusts the unpacked BCD dividend in AX before a division operation.
* Packed Decimal Arithmetic:
  + Packed Decimal integers store two decimal digits per byte.
  + DAA (Decimal Adjust after Addition) Instruction: Converts the binary result of an ADD or ADC operation to packed decimal format.
    - The value to be adjusted must be in AL.
    - If the lower digit is adjusted, the Auxiliary Carry flag is set.
    - If the upper digit is adjusted, the Carry flag is set.
  + DAS (Decimal Adjust after Subtraction) Instruction: Converts the binary result of a SUB or SBB operation to packed decimal format.
    - The value must be in AL.
* Summary:
  + Shift & rotate instructions are some of the best tools of assembly language.
    - Finer control than in high-level languages.
    - SHL, SHR, SAR, ROL, ROR, RCL, RCR.
  + MUL & DIV: Integer operations.
    - Close relatives of SHL & SHR.
    - CBW, CDQ, CWD: Preparation for division.
  + 32-Bit Mode Only:
    - Extended Precision Arithmetic: ADC, SBB.
    - ASCII Decimal Operations: AAA, AAS, AAM, AAD.
    - Packed Decimal Operations: DAA, DAS.

**CHP 8 – Advanced Procedures:**

* Stack Frames:
  + Stack Frame: Also known as an Activation Record.
    - Area of the stack set aside for a procedure’s return address, passed parameters, saved registers, & local variables.
    - Created by the following steps:
      * Calling program pushes arguments on the stack & calls the procedure.
      * The called procedure pushes EBP on the stack, & sets EBP to ESP.
      * If local variables are needed, a constant is subtracted from ESP to make room on the stack.
  + Passing Arguments by Value:
    - Push argument values on stack.
      * Use only 32-bit values in protected mode to keep the stack aligned.
    - Call the called-procedure.
    - Accept a return value in EAX, if any.
    - Remove arguments from the stack if the called-procedure did not remove them.
  + Passing by Reference:
    - Push the offsets of arguments on the stack.
    - Call the procedure.
    - Accept a return value in EAX, if any.
    - Remove arguments from the stack if the called procedure did not remove them.
    - Passing an Array by Reference:
      * The ArrayFill procedure fills an array with 16-bit random integers.
        + ArrayFill can reference an array without knowing the array’s name.
      * The calling program passes the address of the array, along with a count of the number of array elements.
  + RET Instruction:
    - Returns from subroutine.
    - Pops stack into the instruction pointer (EIP or IP). Control transfers to the target address.
    - Syntax: RET, RET n
      * Optional operand n causes n bytes to be added to the stack pointer after EIP (or IP) is assigned a value.
  + Passing 8-Bit & 16-Bit Arguments:
    - Cannot push 8-bit values on stack.
    - Pushing 16-bit operand may cause page fault or ESP alignment problem; incompatible with Windows API functions.
    - Expand smaller arguments into 32-bit values, using MOVZX or MOVSX.
  + Passing Multiword Arguments:
    - Push high-order values on the stack first; work backward in memory.
    - Results in little-endian ordering of data.
  + Saving & Restoring Registers: Push registers on stack just after assigning ESP to EBP; local registers are modified inside the procedure.
  + Local Variables:
    - Only statements within subroutine can view or modify local variables.
    - Storage used by local variables is released when subroutine ends.
    - Local variable name can have the same name as a local variable in another function without creating a name clash.
    - Essential when writing recursive procedures, as well as procedures executed by multiple execution threads.
  + LEA Instruction: Returns offsets of direct & indirect operands.
    - OFFSET operator only returns constant offsets.
    - LEA required when obtaining offsets of stack parameters & local variables.
  + ENTER Instruction: Creates stack frame for a called procedure.
    - Pushes EBP on the stack.
    - Sets EBP to the base of the stack frame.
    - Reserves space for local variables.
  + LEAVE Instruction: Terminates the stack frame for a procedure.
  + LOCAL Directive: Declares a list of local variables.
    - Immediately follows the PROC directive.
    - Each variable is assigned a type.
    - Syntax: LOCAL varlist
  + Non-Doubleword Local Variables:
    - Local variables can be different sizes.
    - How created in the stack by LOCAL directive:
      * 8-Bit: Assigned to next available byte.
      * 16-Bit: Assigned to next even (word) boundary.
      * 32-Bit: Assigned to next doubleword boundary.
  + WriteStackFrame Procedure: Displays contents of current stack frame.
* Recursion:
  + What is Recursion:
    - The process created when…
      * Procedure A calls itself.
      * Procedure A calls procedure B, which in turn calls procedure A.
    - Using a graph in which each node is a procedure & each edge is a procedure call, recursion forms a cycle.
* INVOKE, ADDR, PROC, & PROTO:
  + INVOKE Directive: A powerful replacement for Intel’s CALL instruction that lets you pass multiple arguments.
    - Syntax: INVOKE procedureName [, argumentList].
      * ArgumentList is an optional comma-delimited list of procedure arguments.
      * Arguments can be:
        + Immediate values & integer expressions.
        + Variable names.
        + Address & ADDR expressions.
        + Register names.
  + ADDR Operator: Returns a near or far pointer to a variable, depending on which memory model your program uses:
    - Small Model: Returns 16-bit offset.
    - Large Model: Returns 32-bit segment/offset.
    - Flat Model: Returns 32-bit offset.
  + PROC Directive: Declares a procedure with an optional list of named parameters.
    - Syntax: label PROC paramList.
    - ParamList is a list of parameters separated by commas. Each parameter has the following syntax: paramName : type.
    - Type must either be one of the standard ASM types (BYTE, SBYTE, WORD, etc.), or it can be a pointer to one of these types.
    - Alternate format permits parameter list to be on one or more separate lines, but requires a comma.
    - The parameters can be on the same line, or they can be on separate lines.
  + PROTO Directive: Creates a procedure prototype.
    - Syntax: label PROTO paramList.
    - Parameter list not permitted in 64-bit mode.
    - Every procedure called by the INVOKE directive must have a prototype.
    - A complete procedure definition can also serve as its own prototype.
    - Standard Configuration: PROTO appears at top of the program listing, INVOKE appears in the code segment, & the procedure implementation occurs later in the program.
  + Parameter Classifications:
    - An input parameter is data passed by a calling program to a procedure.
      * The called procedure is not expected to modify the corresponding parameter variable, & even if it does, the modification is confined to the procedure itself.
    - An output parameter is created by passing a pointer to a variable when a procedure is called.
      * The procedure does not use any existing data from the variable, but it fills in a new value before it returns.
    - An input-output parameter is a pointer to a variable containing input that will be both used & modified by the procedure.
      * The variable passed by the calling program is modified.
  + Trouble-Shooting Tips:
    - Save & restore registers when they are modified by a procedure; except a register that returns a function result.
    - When using INVOKE, be careful to pass a pointer to the correct data type.
      * For example, MASM cannot distinguish between a DWORD argument & a PTR BYTE argument.
    - Do not pass an immediate value to a procedure that expects a reference parameter.
      * Dereferencing its address will likely cause a general-protection fault.
* Creating Multimodule Programs:
  + Multimodule Programs: A program whose source code has been divided up into separate ASM files.
    - Each ASM file (module) is assembled into a separate OBJ file.
    - All OBJ files belonging to the same program are linked using the link utility into a single EXE file; this process is called static linking.
  + Advantages:
    - Large programs are easier to write, maintain, & debug when divided into separate source code modules.
    - When changing a line of code, only its enclosing module needs to be assembled again. Linking assembled modules requires little time.
    - A module can be a container for logically related code & data (think object-oriented here…).
      * Encapsulation: Procedures & variables are automatically hidden in a module unless you declare them public.
  + Creating a Multimodule Program:
    - Here are some basic steps to follow when creating a multimodule program:
      * Create the main module.
      * Create a separate source code module for each procedure or set of related procedures.
      * Create an include file that contains procedure prototypes for external procedures (ones that are called between modules).
      * Use the INCLUDE directive to make your procedure prototypes available to each module.
* Summary:
  + Stack parameters
    - More convenient than register parameters.
    - Passed by value or reference.
    - ENTER & LEAVE instructions.
  + Local variables
    - Created on the stack below stack pointer.
    - LOCAL directive.
  + Recursive procedure calls itself.
  + Calling conventions (C, stdcall).
  + MASM procedure-related directives: INVOKE, PROC, PROTO.