Implementing Standard C Control Structures:

**IF THEN ELSE**

```c
if (a == b) {
    mov ax,a
    blah;
}
```

```c
else {
    false clause
}
```

```c
mov ax,a
cmp ax,b
jz doif
```

```c
doelse:...
false clause
... 
jmp endif
```

```c
doiif:...
true clause
... 
jnz doelse
```

```c
doelse:...
false clause
... 
}
```

```c
mov ax,a
cmp ax,b
jnz endif
```

```c
endif:
```

The most complete form of the IF control construct adds additional ELSE IF tests to add more possible choices.

```c
if (a == b) {
    mov ax,a
    first clause
}
```
else if (a == c) {
    second clause
}
else if (a == d) {
    third clause
}
else {
    final clause
}

WHILE

while (a == b) {
    loop body
}

DO WHILE

do {
    loop body
} while (a == b);

ltop: mov ax,a
cmp ax,b
jnz lend
loop body
jmp ltop
lend:
FOR LOOP

for (<init expr>; <test expr>; <reinit expr>) {
    loop body
}

This is semantically equivalent to the following C constructs:

init expr;
while (test expr) {
    loop body
    reinit expr;
}
The implementation of this in assembler follows directly from the above description for the implementation of the WHILE loop.

for (inx = 0; inx < 10; inx++) {movinx, 0

  loop body

ltop:

  mov ax, inx

  cmp ax, 10

  jae lend

  ...

  loop body

  ...

  add inx, 1

  jmp ltop
}

lend:

SWITCH-CASE

The SWITCH construct in C allows for a multi-way decision. An expression is evaluated, and resulting value is used to select a particular block of code to execute from several choices. The C syntax looks like:

switch (expr) {
  case const-expr: statements
  case const-expr: statements
  ...
  default:         statements
}

The value of expr is compared with the value of the const-expr associated with each case statement. If a match occurs, control is transferred to the statements associated with that case. If no match is found, control transfers to the optional default case. If no default case is present, control transfers to the end of the case list. Normally, control will flow from the statements associated with one case into the next unless a break statement is used to cause control to transfer to the end.
This can be implemented in one of two ways in assembler.

In the case where the values for the case statement *const-expr’s are not a contiguous range of values*, it is most efficiently implemented using an IF-THEN-ELSE like construct.

The equivalent C for this would be:

```c
val = expr;
if (val == const_expr1) {
    case 1 statements
}
else if (val == const_expr2) {
    case 2 statements
}
else {
    default statements
}
```

The implementation of this in assembler is exactly as shown above for the IF-THEN-ELSE construct.

**Note:** The C semantics for the switch statement is that control flows through from one case to the next. To prevent this, it is necessary to put `break` statements after each case. If this is done, the semantics of the switch statement is exactly the same as an IF-THEN-ELSE construct.

If the break statements are not included and control flows through from one case to the next, it is necessary to put in explicit jumps to get the proper control flow, and then the semantics are similar to, but not exactly the same as an IF-THEN-ELSE.
Alternatively, if the values of the switch expression are contiguous (e.g. 0, 1, 2, 3) the most efficient implementation is to use a jump table.

In this case, the value of the switch expression is used as the index into a table of pointers to the blocks of statements associated with each case. An indirect jump using the indexed location in the jump table is used to transfer control to the appropriate statement block.

**Note:** When using a jump table, it is important to check that the switch expression value is within the range of values allowed for the jump table. If an out of range value is used to index into the jump table, the program will crash.
The jump table can be placed either in the code segment or the data segment. A code segment implementation will be illustrated:

```asm
switch (inx) {
    case 0:
        case 0 body
        break;
    case 1:
        case 1 body
        break;
    case 2:
        case 2 body
        break;
    default:
        default body
}
```

```
    mov     si, inx
    cmp     si, 0
    jb      ldefault
    cmp     si, 3
    ja      ldefault
    add     si, si
    jmp     jmptab[si]

    jmptab dw lcase0
    dw lcase1
    dw lcase2

    lcase0:...
    case 0 body
    ...
    jmp     lend

    lcase1:...
    case 1 body
    ...
    jmp     lend

    lcase2:...
    case 2 body
    ...
    jmp     lend

    ldefault ...
    default body
    ...

    lend:
```

Note: In the case where the value of the first case expression isn’t 0, it is necessary to subtract that value from the switch value before computing the index into the jump table.
One of the advantages of programming in assembly language is that there is a great deal more flexibility available in implementing the flow of control in the program.

It is not necessary to implement the assembler code exactly as shown above for the C language constructs. However, it is also much easier in assembler to create buggy, impossible to understand code by undisciplined coding.

It is generally advisable to use a disciplined, structured approach to implementing the flow of control in an assembler program and to use structures similar to those shown above.

**Interrupt Service Routines**

The interrupt service routine is the subroutine that handles the interrupt. Its address is stored in the interrupt vector table.

When an interrupt occurs, the CPU could be doing anything, and could have values in any register or flag that are important. The Interrupt Service routine, ISR, has no way of knowing what the interrupted code was doing when the interrupt occurred, or which registers are important, so it must preserve all CPU state.

- The first thing that most ISR’s do is to push all registers that they are going to modify, to preserve their values. Note that the FLAGS register was automatically pushed onto the stack before control was transferred to the ISR, so it isn’t necessary to push the flags.
- The ISR then performs whatever tasks are necessary to service the device that signaled the interrupt.
- After servicing the device, the ISR will restore the registers by popping their values from the stack and then perform an IRET instruction to return to the interrupted task.
Assume, for example, that the interrupt was generated by a character-oriented device, such as a serial port. A device like a serial port will have both an input stream and an output stream of characters. As each character is received coming into the port, an interrupt will be generated. The interrupt service routine will read the character from the input register of the serial port, and typically will store the character into an input queue or circular buffer. The higher-level program that is consuming characters from the serial port will read them from the input buffer when they become available. By using an interrupt driven input queue, the consuming application is freed from the necessity of always having to be able to service the serial port before a character is lost. Similarly, on the output side, the application will typically write the output characters into an output queue. The serial port hardware will generate an interrupt whenever the output buffer register on the serial port is empty and ready to send another character. The interrupt service routine will read the next character from the output queue and write it to the output buffer of the serial port hardware.
Debugger Trace and Breakpoint Interrupts

There are two features of the interrupt system on the 8088/8086 that are intended specifically to support the writing of debuggers. The first is the single step interrupt or trace interrupt (INT 1). The second is the breakpoint interrupt (INT 3).

The single step or trace interrupt mechanism allows the debugger to easily execute a single instruction in the debug child program and then return control to the debugger. When the trace flag (TF) is set in the FLAGS register, the CPU will execute one instruction and then perform an INT 1 operation, as if an INT 1 instruction had been executed. The debugger will hook into the INT 1 vector and point it at its single step entry point. To execute the child program, the debugger will set up all of the child program’s registers and place an IRET frame on the child’s stack, with the FLAGS entry on the stack having the trace flag set. When the debugger performs an IRET, control transfers to the child program and executes one instruction. After executing that instruction, the CPU performs the INT 1, returning control to the debugger, which then saves the child registers into its local state variables, switches to its own stack and register contents and then returns to its own internal operation.

The breakpoint interrupt works in a similar manner, in that the interrupt mechanism is used to return control to the debugger after executing some number of child program instructions. The INT 3 instruction is a special one byte version of the general INT instruction. The breakpoint opcode is 0CCh. The debugger can set a breakpoint by saving the first opcode byte of the instruction where the breakpoint is to occur and then replacing that byte with the 0CCh opcode. The debugger returns control to the child program again by building an IRET frame on the stack, restoring the child registers and then performing an IRET to resume execution of the child’s code. When the breakpoint instruction is executed, the CPU will perform an INT 3 operation. The debugger will have hooked the INT 3 vector with the address of its breakpoint entry point. The breakpoint entry point will save all of the child’s registers, switch to the debugger stack, restore the original opcode byte that the breakpoint had replaced, and then resume the internal operation of the debugger.
It is important that the breakpoint instruction be a single byte instruction. If the breakpoint required multiple bytes, it could overlap two or more instructions, and make it impossible to reliably set a breakpoint on the desired instruction in some cases.