Program Control Instructions

- Generally Modify **CS:IP**
- Causes Modification in Execution Sequence (of Instructions)
- When Such a Program Flow Change Occurs:
  - a) Instructions in the BIU Inst. Queue Become Invalid
  - b) BIU Directly Fetches **CS:IP** Instruction from Memory
  - c) While EU Executes New Instruction, BIU Flushes/Refills Inst. Queue
- Classification
  - a) Jumps - Unconditional Control Transfers (synchronous)
  - b) Branches - Conditional Control Transfer
  - c) Interrrupts - Unconditional Control Transfers (asynchronous)
  - d) Iteration - More Complex Type of Branch

Control Instruction Summary

UNCONDITIONAL

```
jmp     LABEL           ;next instruction executed has LABEL
call    LABEL           ;next instruction executed has LABEL
ret                     ;next instruction executed is after the call
hlt                     ;nothing executed until RESET signal
```

ITERATION

```
loop    LABEL           ;cx <-- cx - 1, jump to LABEL if cx > 0
loope/loopz  LABEL      ;same as loop but ZF=1 also required
loopne/loopnz           ;same as loop but ZF=0 also required
```

INTERRUPTS

```
int     <immed8>        ;Invoke the int. handler specified by immed8
into    <immed8>        ;same as int but OF=1 also
iret                    ;Return from interrupt handler
```

CONDITIONAL to follow

Simplest Control Instruction, **jmp**

```
jmp     LABEL   ;LABEL is offset address of instruction
                ;in the code segment
```

THIS IS EQUIVALENT TO:

```
mov     ip,     OFFSET LABEL
```

The mov form is *illegal* since it is not allowed to do this inside the code segment

3 Forms of **jmp**

SHORT - 2 bytes, allows jump to ±127 locations from current address

EB	disp

NEAR - 3 bytes, allows jump to ±32K locations from current address

E9	disphi	displo

FAR - 5 bytes anywhere in memory

EA	IP lo	IP hi	CS lo	CS hi

386+ - Must Far Jump in to Enter Protected Mode!!!

Example with Short Jump

```
;Causes bx to count by 1 from 0 to 65535 to 0 to 65535 to …
        xor     bx,     bx      ;Clear bx and initialize status flags
start:  mov     ax,     1       ;ax <-- 1
        add     ax,     bx      ;ax <-- ax+bx
        jmp     next            ;add a displacement to IP
                                ;       (+2 from xor to mov)
        xor     bx,     bx      ;Clear bx and initialize flags
        xor     ax,     ax      ;Clear ax and initialize flags
next:   mov     bx,     ax      ;bx <-- ax
        jmp     start           ;add a displacement to IP
                                ;       (a negative value - 2's comp.)
```

Indirect Jump

- Address of Target is in Register
- Does ***NOT*** Add disp to **IP** - Transfer **REG** Contents to **IP**

```
;assume that si contains either 0, 1 or 2
        add     si,     si      ;si <-- 2*si
        add     si,     OFFSET TABLE     ;si <-- si + <address of TABLE>
        mov     ax,     cs:[si]  ;ax gets an address from the jump table
        jmp     ax              ;ip <-- ax
;the following jump TABLE is defined in the code segment!!!!
TABLE:  DW      ZERO
        DW      ONE
        DW      TWO
ZERO:   ;code for ZERO option
                .
                .
                .
ONE:    ;code for ONE option
                .
                .
                .
TWO:    ;code for TWO option
                .
                .
                .
```

Indirect Addressed Jump

- Address of Target is in Register
- Does ***NOT*** Add disp to **IP** - Transfer **MEM** Contents to **IP**

```
;assume that si contains either 0, 1 or 2
        add     si,     si      ;si <-- 2*si
        add     si,     OFFSET TABLE     ;si <-- si + <address of TABLE>
        jmp     cs:[si]         ;ip gets an address from the jump table
;the following jump TABLE is defined in the code segment!!!!
TABLE:  DW      ZERO
        DW      ONE
        DW      TWO
ZERO:   ;code for ZERO option
                .
                .
                .
ONE:    ;code for ONE option
                .
                .
                .
TWO:    ;code for TWO option
                .
                .
                .
```

Conditional Control Instruction Summary
Simple Flag Branches

Jump based on single flag

CONDITIONAL

```
jc      LABEL   ;jump on carry (CF=1)
jnc     LABEL   ;jump on no carry (CF=0)
je/jz   LABEL   ;jump if ZF=1 - jump if equal/zero
jne/jnz LABEL   ;jump if ZF=0 - jump not equal/jump if zero|
jo      LABEL   ;jump if OF=1 - jump on overflow
jno     LABEL   ;jump if OF=0 - jump if no overflow
js      LABEL   ;jump on sign flag set (SF=1)
jns             ;jump if no sign flag (SF=0)
jp/jpe          ;jump if PF=1 - jump on parity/parity even
jnp/jpo LABEL   ;jump if PF=0 - jump on no parity/parity odd
```

Conditional Control Instruction Summary
Branches for unsigned comparisons

Jump is based on flags used for unsigned number comparison (based on C, Z flag)

CONDITIONAL

```
ja/jnbe LABEL   ;jump if CF=ZF=0 - jump above-jump not below/equal
jae/jnb LABEL   ;jump if CF=0 - jump above/equal-jump not below
jb/jnae LABEL   ;jump if CF=1 - jump below-jump not above/equal
jbe/jna LABEL   ;jump if CF=1 or ZF=1 - jump equal - jump zero
```

Typical use:

```
cmp  al,bl
jb   there     ; jump if al is 'below' bl
               ; unsigned comparison
```

Conditional Control Instruction Summary
Branches for signed comparisons

Jump is based on flags used for signed number comparison (based on Z, S, V flags)

CONDITIONAL

```
jg/jnle LABEL   ;jump if ZF=0 and (SF=OF) - jump greater/not less
nor equal
jge/jnl LABEL   ;jump if SF=OF - jump greater-equal/not less than
jl/jnge LABEL   ;jump if SF ≠ OF - jump less than/not greater nor
equal
jle/jng LABEL   ;jump if ZF=1 or SF ≠ OF - jump less or equal/not
great than jnc LABEL   ;same as jae/jnb
```

Typical use:

```
cmp  al,bl
jl   there     ; jump if al is less than bl
               ; signed comparison
```

Branch - Conditional Transfers

- **Always SHORT jumps in 86-286**
- Can be SHORT or NEAR in 386+
- Condition Tested is Content of SF, ZF, CF, PF, OF

CONDITIONAL SET 386+

- Result in Byte of Memory - Either 00h or 01h
- Useful for Saving Flag Contents in Memory

EXAMPLE

```
setb    T1      ;T1 <-- 01h if CF=1 else T1 <-- 00h

seto    T1      ;T1 <-- 01h if OF=1 else T1 <-- 00h
```

Iteration Instruction, **loop**

- Combination of Decrement **cx** and Conditional Jump
- Decrements **cx** and if **cx≠0** jumps to LABEL
- 386+ **loopw** (**cx** operation) and **loopd** (**ecx** operation)

Example:

```
ADDS    PROC    NEAR
        mov     cx,     100
        mov     si,     OFFSET BLOCK1
        mov     di,     OFFSET BLOCK2
        cld
AGAIN:  mov     bx,     di
        lodsw
        add     ax,     [bx]
        mov     di,     bx
        stosw
        loop    AGAIN
        ret
ADDS    ENDP
```

Iteration Instruction, **loop**

```
ADDS    PROC    NEAR
        mov     cx,     100     ;cx <-- 64h - number of words to add
        mov     si,     OFFSET BLOCK1  ;si <-- offset of BLOCK1 (in ds)
        mov     di,     OFFSET BLOCK2  ;di <-- offset of BLOCK2 (in ds)
        cld                     ;Auto-increment si and di, DF=0
AGAIN:  mov     bx,     di      ;bx <-- di, save offset of BLOCK2
        lodsw                   ;ax <-- ds:[si], si<--si+2, di<--di+2
        add     ax,     [bx]    ;ax <-- ax + ds:[bx]
        mov     di,     bx      ;di <-- bx, restore di with
                                ;   offset in BLOCK2
        stosw                   ;es:[di] <-- ax, si<--si+2, di<--di+2
        loop    AGAIN           ;cx <-- cx - 1, if cx≠0 jump to AGAIN
        ret                     ;ip <-- ss:[sp]
ADDS    ENDP
```

Procedures

- Group of Instructions that Perform Single Task
  - *(can be used as)* a *SUBROUTINE*

`call` - invokes subroutine - pushes **`ip`**
`ret` - returns from subroutine - pops **`ip`**

- Uses MASM Directives: **`PROC`** and **`ENDP`**
- Must Specify

`NEAR` - intrasegment
`FAR` - intersegment

- Difference is op-code of **`ret`**

`NEAR - c3h - pops IP`
`FAR - cbh - pops CS, pops IP`

`call` Instruction

- Differs from **`jmp`** Since Return Address on Stack

`NEAR` call: 3 bytes - 1 opcode and 2 for **`IP`**
`FAR` call: 5 bytes - 1 opcode, 2 for **`IP`** and 2 for **`CS`**

- **`call`** with operand - can use 16-bit offset in any register except segment registers

```
call    bx    ;pushes ip then jumps to cs:[bx]
```

`call` Instruction - Example

```
        mov    si,    OFFSET COMP
        call   si

               .
               .
               .
COMP    PROC   NEAR
        push   dx
        mov    dx, 03f8h
        in     al,    dx
        inc    dx
        out    dx,    al
        pop    dx
        ret
COMP    ENDP
```

`call` Instruction - Example Explained

```
        mov     si,     OFFSET COMP     ;get offset of COMP subroutine
        call    si                      ;push ip, ip<--si
                .
                .
                .
COMP    PROC    NEAR
        push    dx              ;Save current contents of dx
        mov     dx, 03f8h       ;dx <-- 03f8h (an immediate data Xfer)
        in      al,     dx      ;al receives 1 byte of data from I/O
                                ; device with output port address 03f8h
        inc     dx              ;dx<--03f9h
        out     dx,     al      ;send 1 byte of data to I/O device
                                ; input port with address 03f9h
        pop     dx              ;restore dx to value at call time
        ret                     ;ip<--ss:[sp], sp<--sp+2
COMP    ENDP
```

`call` Instruction with Indirect Address

- Useful for Choosing Different Subroutines at Runtime
- Can Use a Table (like the jump Table Example)

```
                ;Assume bx contains 1, 2 or 3 for subroutine desired
                TABLE   DW      ONE
                        DW      TWO
                        DW      THREE
                        dec     bx
                        add     bx,     bx
                        mov     di, OFFSET TABLE
                        call    cs:[bx+di]
                        jmp     CONT
                ONE     PROC    NEAR
                        …
                ONE     ENDP
                TWO     PROC    NEAR
                        …
                TWO     ENDP
                THREE   PROC    NEAR
                        …
                THREE   ENDP
                CONT:   nop
```

`call` Instruction with Indirect Address

```
;Table of addresses of subroutines
TABLE   DW      ONE
        DW      TWO
        DW      THREE
;bx contains 1, 2 or 3 - desired subroutine
        dec     bx              ;bx <-- 0, 1 or 2
        add     bx,     bx      ;bx <-- 0, 2 or 4
        mov     di, OFFSET TABLE        ;di <-- TABLE offset
        call    cs:[bx+di]      ;push ip, ip<--offset of subroutine
        jmp     CONT            ;ip <-- offset of nop instruction
ONE     PROC    NEAR
        …
ONE     ENDP
TWO     PROC    NEAR
        …
TWO     ENDP
THREE   PROC    NEAR
        …
THREE   ENDP
CONT:   nop
```

`ret` Instruction

NEAR - pops 16-bit value places in **IP**
FAR - pops 32-bit value places in **CS:IP**

- Type is Determined by **PROC** Directive
- Other Form of ret has Immediate Operand
  a) modifies SP before restoring IP
  b) Useful When Subroutine Modifies Stack

EXAMPLE

```
TEST  PROC  NEAR
      push  ax    ;ss:[sp]<--ax, sp<--sp-2
      push  bx    ;ss:[sp]<--ax, sp<--sp-2
            …
      ret   4     ;ip<--ss:[sp+4]
TEST  ENDP
```