5. 주로 사용하는 명령어

5.1 데이터 전송 명령어
5.2 8086 주소지정 명령어
5.3 펜티엄 주소지정 명령어
5.4 스택 명령어
5.5 산술논리 명령어
5.6 이동, 회전 명령어
5.7 곱셈, 나눗셈 명령어
5.8 제어 명령어
5.9 서브프로그램 명령어

5.1 데이터 전송 명령어

- mov 명령어
- mov 연산자는 피연산자의 크기가 같다.
- 피연산자가 모두 메모리인 경우를 허용하지 않는다.

[라벨:]	MOV	레지스터/메모리, 레지스터/메모리/직접 값

- 레지스터 이동
- mov edx, ecx ; 레지스터에서 레지스터 이동
- mov es, ax ; 레지스터에서 세그먼트 레지스터 이동
- mov [bytefield], dh; 레지스터에서 메모리 이동, 직접메모리주소
- mov [di], bx ; 레지스터에서 메모리 이동, 간접메모리주소

5.1 데이터 전송 명령어

- 직접 데이터 이동
- mov cx, 40h ; 직접 데이터를 레지스터로 이동
- mov [bytefield], 25 ; 직접 데이터를 메모리로 이동
- 직접 메모리 이동
- mov ch, [bytefield] ; 메모리에서 레지스터로 이동
- 세그먼트 레지스터 이동
- mov ax, ds ; 세그먼트 레지스터에서 레지스터로 이동
- mov [wordfield], ds ; 세그먼트 레지스터에서 메모리로 이동
- 올바르지 않는 MOV 연산은 다음과 같다.
- mov al, si ; 레지스터의 길이가 같이 않음
- mov [byte_val1], [byte_val2] ; 메모리에서 메모리로 이동
- mov es, 255 ; 직접 값을 세그먼트 레지스터로 이동
- mov es, ds ; 세그먼트 레지스터에서 세그먼트 레지스터

5.1 데이터 전송 명령어

- movzx와 movsx 명령어
- movzx와 movsx 연산자는 바이트는 워드로 워드는 2중 워드로 확장하여 전송.

[라벨:]	MOVZX/MOVSX	레지스터/메모리, 레지스터/메모리/직접 값

  - movzx는 상위 비트를 0으로 채움.
  - movsx는 상위 비트를 sign 비트로 채움.
  - al = 10110000
  - movzx cx, al ; 00000000 10110000
  - movsx cx, al ; 11111111 10110000

5.1 데이터 전송 명령어

- xchg 명령어
- xchg 연산자는 2개의 데이터 항목을 교환.

[라벨:]	xchg	레지스터/메모리, 레지스터/메모리

- 두 레지스터 사이의 데이터 교환.
- 레지스터와 메모리 사이의 데이터 교환.

```
wordq    dw      35h     ; 워드 데이터 항목
         ...
   xchg  cl, bh          ; 두 레지스터의 내용을 교환
   xchg  cx, [wordq]     ; 레지스터와 메모리의 내용을 교환
```

5.2 8086 주소 지정 방식

- 즉치(Immediate) 주소지정 방식
  - 이 주소지정 방식에서 피연산자 필드의 내용은 상수 값이나 상수 식의 결과 값을 가진다. 이 주소지정 방식은 세그먼트 레지스터와 플래그 레지스터를 제외한 모든 레지스터에 데이터를 적재하기 위해 이용될 수 있다.

```
byte_dt    db 0                    ; 바이트를 정의
word_dt    dw 0                    ; 워드를 정의
dword_dt   dd 0                    ; 더블워드를 정의
              ...
    mov    byte [byte_dt], 50      ; 즉시 값을 메모리에(바이트)
    mov    word [word_dt], 40h     ; 즉시 값을 메모리에(워드)
    mov    dword [dword_dt], 0     ; 즉시 값을 메모리에(더블워드)
    mov    ax, 0245h               ; 즉시 값을 레지스터에(워드)
```

5.2 8086 주소 지정 방식

- 즉치(Immediate) 주소지정 방식
  - mov al, 0245h ; 올바르지 않은 즉시 값의 길이
  - mov al, 48h ; 올바른 즉시 값의 길이

  - 세그먼트 레지스터에 어떤 값을 적재하기 위해서는 먼저 이 값을 범용 레지스터에 적재한 다음, 이 범용 레지스터에 적재된 값을 세그먼트 레지스터로 복사해야 한다.
  - mov ax, 2550h
  - mov ds, ax
  - 만약, 다음과 같이 작성한다면 오류를 발생시킨다.
  - mov ds, 0123h ; 오류 발생

5.2 8086 주소 지정 방식

- 레지스터 주소지정 방식
  - 이 주소지정 방식은 처리할 데이터를 레지스터에 적재한 후 명령어에서 레지스터를 피연산자로 지정하는 방식이다. 8, 16, 32비트 크기의 레지스터를 지정할 수 있다. 명령어에 따라서 레지스터가 첫 번째 피연산자, 두 번째 피연산자, 또는 두 개의 피연산자 모두에 올 수 있다. 이 방식은 어떠한 메모리 참조도 포함하지 않기 때문에 가장 빠른 연산의 유형이다.
  - `mov bx, dx ; dx의 내용을 bx에 복사`
  - `mov es, ax ; ax의 내용을 es에 복사`
  - `add al, bh ; bh의 내용을 al에 더함`
  - 이 주소지정 방식에서 출발지와 목적지의 데이터 크기는 같아야 한다. 아래에 보인 예와 같이 출발지는 16비트이고 목적지가 8비트이면 오류를 발생시킨다.
  - `mov cl, ax ; 오류 발생`

5.2 8086 주소 지정 방식

- 직접 메모리 주소지정 방식
  - 이 주소지정 방식에서는 메모리 위치의 오프셋을 지정한다. 물리 주소는 [세그먼트 레지스터 : 오프셋] 형식으로 지정한다. 여기서 세그먼트 레지스터는 기본으로 DS(데이터 세그먼트)를 사용한다. 오프셋은 세그먼트 레지스터에 저장되어 있는 세그먼트의 시작 주소로부터 피연산자가 저장되어 있는 곳까지의 거리를 나타내며, 다른 말로 변위(displacement)라고 한다.
  - add [byte_dt], dl ; 레지스터를 메모리에 더함(바이트)
  - mov bx, [word_dt] ; 메모리를 레지스터로 이동(워드)

  - DS 이외의 세그먼트를 사용하는 경우에는 세그먼트를 표시해주어야 한다.

  - add [es:byte_dt], dl ; ES를 사용하여 물리 주소를 계산

5.2 8086 주소 지정 방식

- 간접 메모리 주소지정 방식
  - [세그먼트 레지스터 : 오프셋] 형식으로 주소를 지정하는 방식이다. 이러한 목적을 위해서 사용하는 레지스터는 베이스 레지스터(base register, BX와 BP)와 인덱스 레지스터(index register, DI와 SI)이고, 대괄호([]) 안에 표현된다.
  - [di]와 같이 나타내는 간접 메모리 주소는 프로그램이 실행될 때 사용할 메모리 주소가 DI에 있다는 것을 어셈블러에게 알린다. BX, DI, SI는 데이터 세그먼트에 속한 데이터를 처리하기 위해서 사용되며, ds:bx, ds:di, ds:si와 같이 ds와 연관된다. BP는 스택에 속한 데이터를 처리하기 위해서 사용되며, SS와 연관된다.
  - 첫 번째 피연산자가 간접 메모리 주소이면, 두 번째 피연산자는 레지스터나 즉시 값을 참조한다. 두 번째 피연산자가 간접 메모리 주소이면, 첫 번째 피연산자는 레지스터를 참조한다. 대괄호 안에 표현된 BP, BX, DI, SI 피연산자는 간접 메모리 연산자를 의미하고, 프로세서는 프로그램이 실행중일 때 레지스터의 내용을 오프셋 주소로 다룬다.

5.2 8086 주소 지정 방식

- 간접 메모리 주소지정 방식
  - data db 50 ; 바이트를 정의
  - ...
  - mov bx, data ; 오프셋을 BX에 저장
  - mov [bx], cl ; CL을 data로 이동

  - 이 예제는 MOV 명령이 먼저 BX를 data의 오프셋 주소로 초기화를 한다. 다음에 MOV가 BX의 주소를 사용하여 이 주소가 가리키는 메모리 위치에 CL을 저장한다.

5.2 8086 주소 지정 방식

- 인덱스/베이스 변위 주소지정 방식
  - 이 주소지정 방식은 베이스 레지스터인 BX와 BP, 인덱스 레지스터인 DI와 SI를 사용하여 주소를 지정하는 방식이다. 이 방식은 실제 주소(effective address)를 생성하기 위해서 변위와 결합한다. 다음 예의 MOV 명령어는 data의 시작으로부터 2바이트 떨어진 위치로 0을 이동시킨다.
  - data resb 365 ; 바이트를 정의
  - ...
  - mov bx, data ; BX에 오프셋을 적재
  - mov byte [bx+2], 0 ; data+2에 0을 이동

5.2 8086 주소 지정 방식

- 베이스-인덱스 주소지정 방식(2차원 배열)
  - 이 주소지정 방식은 실제 주소를 생성하기 위해서 베이스 레지스터와 인덱스 레지스터를 결합한다. 예를 들어, [BX+DI]는 BX의 주소와 DI의 주소를 더하는 것을 의미한다. 이 주소지정 방식은 2차원 배열을 주소지정하는 데 일반적으로 사용된다. 예를 들면, BX는 행을 참조하고 DI는 열을 참조한다.
  - mov ax, [bx+si] ; 메모리 워드를 이동
  - add [bx+di], cl ; 바이트를 메모리에 더함

5.2 8086 주소 지정 방식

- 변위를 갖는 베이스-인덱스 주소지정 방식
  - 이 주소지정 방식은 베이스-인덱스 주소지정 방식의 변형이다. 이 주소지정 방식은 실제 주소를 생성하기 위해서 베이스 레지스터, 인덱스 레지스터, 변위를 결합한다.
  - mov ax, [bx+di+10]
  - mov cl, [bx+di+data]

5.3 펜티엄(Pentium) 주소 지정 방식

- 펜티엄 프로세서에서 16비트 레지스터를 사용한 주소지정 방식은 8086과 동일.
- [base register] + [index register] + [offset]
- 베이스 레지스터는 BP 또는 BX이다. BP 레지스터를 사용하면 스택 세그먼트(SS)를 사용하며, 그 외는 데이터 세그먼트를 사용한다. 인덱스 레지스터는 SI 또는 DI이다. 오프셋은 8비트 또는 16비트 길이이다. 베이스 레지스터, 인덱스 레지스터와 오프셋은 각기 생략할 수 있다.
- 80386 이상 펜티엄에서는 32비트 레지스터를 사용하여 다음과 같이 주소를 지정할 수 있다.
- [base register] + [index register]*factor +[offset]
- 베이스 레지스터는 EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP이다. ESP와 EBP를 사용하면 스택 세그먼트(SS)를 사용하며, 그 외는 데이터 세그먼트를 사용한다. 인덱스 레지스터는 ESP를 제외하고 EAX, EBX, ECX, EDX, ESI, EDI, EBP가 될 수 있다. Factor는 인덱스 레지스터에 곱하는 상수로 1, 2, 4, 8 중의 하나이다. 오프셋은 8비트, 16비트 또는 32비트 길이다. 32비트 레지스터와 16비트 레지스터를 혼용해서 사용할 수 없다.

5.3 펜티엄(Pentium) 주소 지정 방식

- mov ax, [edx + ecx*2 + 2000h]
- 상기 예는 ECX 레지스터에 2를 곱하고, 그에 EDX 레지스터의 값과 2000H를 더한 값이 메모리의 오프셋이 된다.
- 다음 예는 메모리 주소지정의 예이다.
- mov al, [dx] ; 오류. DX는 인덱스 또는 베이스 레지스터가 아니다.
- mov al, [edx] ; 맞음
- mov al, [ebx+di] ;오류.16비트와 32비트 레지스터를
  ;혼용 있음.
- mov eax, [si+5] ; 맞음
- mov ebx, [cs:eax + edx*4 + 10] ; 맞음

5.4 스택 명령어

- 스택은 주로 데이터를 임시로 저장하기 위해서 사용.
- 컴퓨터 시스템은 사용할 수 있는 레지스터의 수가 한정되어 있기 때문에 어떤 레지스터를 중복해서 사용해야 할 경우가 생긴다. 이때 중복해서 사용되는 레지스터에 이전에 저장되었던 데이터를 임시로 스택에 저장한다. 또한 카운터 레지스터인 경우에는 중복해서 사용되는 경우가 발생한다. 이러한 경우에 레지스터에 있던 내용을 스택에 임시로 저장했다가 뒤에 복원하여 사용한다. 메모리의 내용을 임시로 저장하기 위해서도 스택이 사용된다.
- 이외에도 스택은 프로시저 호출 사이에 매개변수를 전달하기 위해서 사용된다. 매크로 명령을 사용하지 않는 한, 어셈블리 프로그램에서는 매개변수를 직접 표현할 수 없다. 따라서 한 프로시저가 다른 프로시저를 호출하면서 필요한 데이터를 넘겨주고 필요한 자료를 넘겨받기 위해 스택을 사용한다.

5.4 스택 명령어

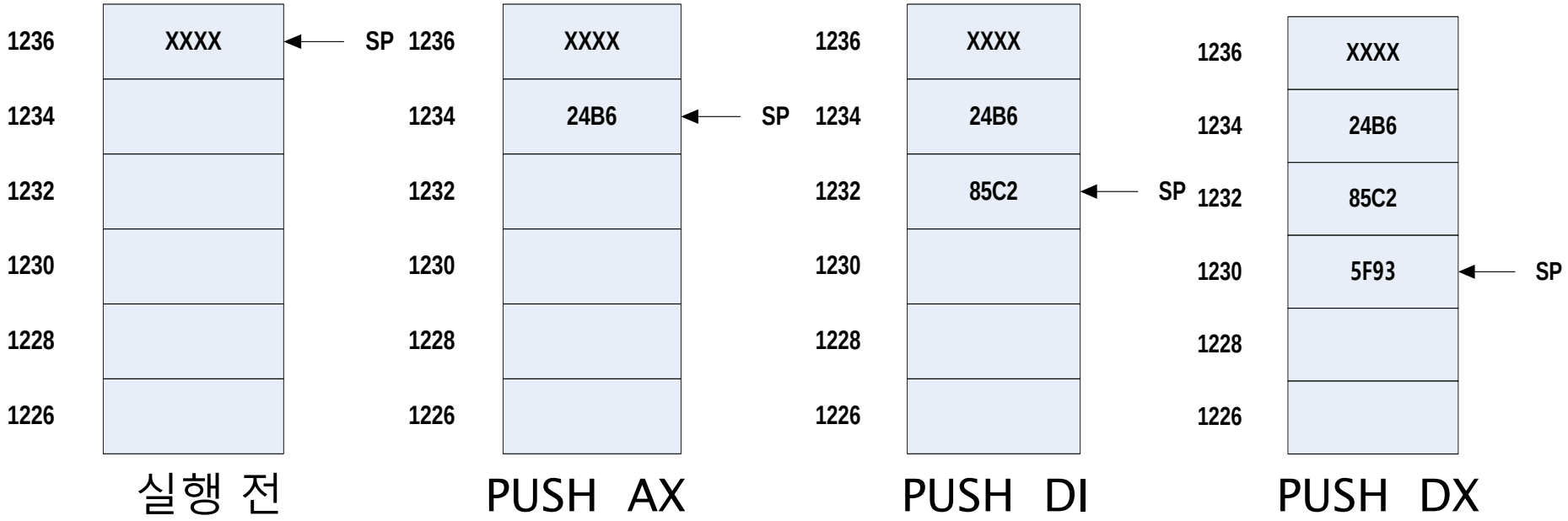
- PUSH 명령어
  - PUSH는 스택에 데이터를 저장하는 명령어이다.

[라벨:]	PUSH	레지스터/메모리

  - PUSH 명령어의 피연산자는 레지스터나 메모리가 될 수 있으나, 피연산자의 크기는 워드 또는 더블 워드이어야 한다.
  - 명령을 실행하면 스택 포인터(SP)를 2만큼 감소시키고 스택 포인터가 가리키는 곳인 스택의 최상위에 피연산자의 내용을 저장한다.

5.4 스택 명령어

- 현재 AX에 24B6H, DI에 85C2H, DX에 5F93H가 있고, SP가 1236H라고 할 때, PUSH 명령 실행에 따른 스택의 상태를 그림에 나타내었다. PUSH 명령을 실행하기 전의 스택은 다음과 같다.



PUSH EAX 명령은 더블 워드 레지스터인 EAX를 스택에 저장한다. 이를 위해서 스택 포인터가 4 감소한 후에 EAX를 저장한다.

5.4 스택 명령어

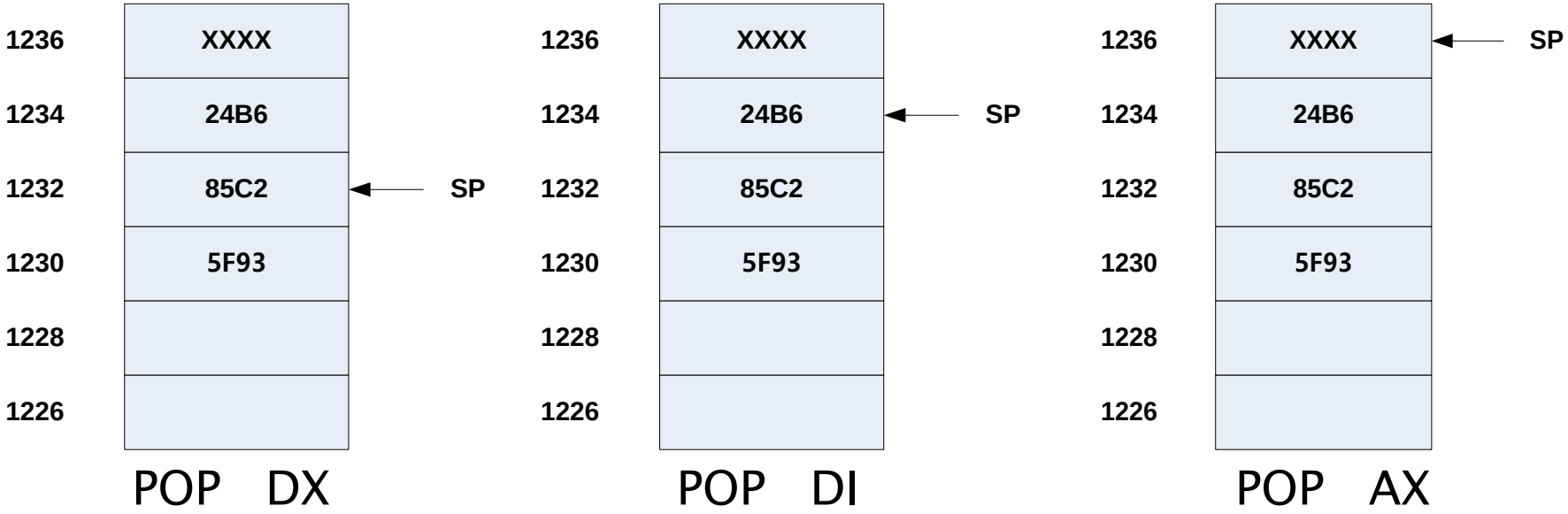
- POP 명령어
  - POP 명령어는 스택으로부터 데이터를 꺼내기 위해 사용된다.

[라벨:]	POP	레지스터/메모리

  - POP 명령어의 피연산자는 레지스터나 메모리가 될 수 있으나, 피연산자의 크기는 워드 또는 더블 워드이어야 한다.
  - 스택 포인터가 가리키는 스택의 최상위에서 데이터를 꺼내 피연산자에 적재하고, 스택 포인터를 2만큼 증가시킨다.

5.4 스택 명령어

- 스택 포인터는 스택의 최상위 주소를 가리킨다. 다음에 POP 명령의 예를 나타내었고, 실행에 따른 스택의 상태를 그림에 보였다.



- POP 명령을 실행하면 데이터가 인출되고 나서 삭제되는 것이 아니라 스택에 그대로 있게 된다. 스택에 들어간 데이터가 삭제되는 것은 나중에 그 자리에 다른 데이터가 들어올 때이다.

5.5 산술 논리 명령어

- ADD 명령어

[라벨:]	ADD	레지스터/메모리, 레지스터/메모리/즉시 값

  - ADD 명령어는 첫 번째 피연산자의 값을 두 번째 피연산자의 값과 더한 후 결과 값을 첫 번째 피연산자에 저장한다.
  - 첫 번째 피연산자는 레지스터나 메모리가 될 수 있고, 두 번째 피연산자는 레지스터, 메모리, 그리고 즉시 값이 될 수도 있다.
  - ADD 명령어에서 반드시 지켜야할 것은 두 피연산자 모두 메모리가 될 수 없다는 것과 두 피연산자의 크기는 같아야한다는 것이다.
  - ADD 연산은 플래그 레지스터의 오버플로, 캐리, 제로, 사인 플래그를 변화시킨다.

5.5 산술 논리 명령어

- 오버블로(overflow)
- 2의 보수 연산에서 산술 연산은 수의 범위를 초과할 수 있다.
- 예를 들면 AL 레지스터에 60H가 저장되어 있을 경우, 덧셈 명령어 'ADD AL, 20H'는 AL 레지스터에 80H의 결과 값을 생성한다. 2의 보수에서 60H와 20H는 양수로 각각 10진수 96과 32이다. 이 두 수를 더하면 80H가 된다. 80H를 2의 보수로 보아서 10진수로 변환하면 -128이다.
- 2의 보수 연산에서 양수와 양수를 더하거나 음수와 음수를 더하는 경우에 오버플로가 발생할 수 있으며, 오버플로가 발생하면 플래그 레지스터의 오버플로 플래그가 '1'이 된다.

5.5 산술 논리 명령어

- 바이트를 워드로 확장(CBW : Convert byte to word)
- AL 레지스터에서 수행되는 덧셈 연산이 오버플로 때문에 발생되는 오류를 방지하기 위해서는 AX 레지스터에서 연산이 수행되도록 하는 것이다. 이러한 목적으로 사용되는 명령어가 CBW이다.
- 이 명령어는 AL 레지스터에 있는 값의 부호 비트(0 또는 1)를 자동으로 AH 레지스터 전체에 전달해 주는 역할을 한다.

[라벨:]	CBW	

- 이 명령어는 피연산자가 없으며, AX 레지스터만 사용할 수 있다.

```
                                        AH      AL
...                                     xx      60h
cbw         ; AL의 부호를 AH로 확장      00      60
add  ax, 20h ; AX에 더함                00      80
```

5.5 산술 논리 명령어

- 워드를 2중 워드로 확장(CWD : Convert word to double word)
- 1워드 수의 부호 비트를 확장하는데 사용하는 명령어는 CWD이다.

[라벨:]	CWD	

- 이 명령어도 피연산자가 없으며, AX 레지스터에 있는 값의 부호 비트를 DX 레지스터에 복사하여 'DX:AX'로 워드 크기의 수를 더블워드로 확장한다. 이 명령어를 사용할 때 유의할 사항은 DX:AX만 사용한다는 것이다.
- mov ax, [word1] ; 워드를 AX로 이동
- cwd ; 워드를 DX:AX로 확장

5.5 산술 논리 명령어

- ADC 명령어

[라벨:]	ADC	레지스터/메모리, 레지스터/메모리/즉시 값

  - ADC(Add with Carry) 명령은 덧셈을 수행하여 발생한 캐리의 영향을 고려하는 연산이다.
  - '더블워드 + 더블워드' 연산을 수행할 때 워드 크기로 나누어서 덧셈을 수행한다. 첫 번째 워드 + 워드를 수행하여 레지스터의 크기를 초과하여 캐리가 발생하면 캐리 플래그가 1로 설정된다. 이 때, 그 다음 '워드 + 워드' 덧셈에서 앞에서 발생한 캐리의 영향을 포함하는 것이 이 연산이다. 즉, 캐리 플래그와 함께 더하는 것이다.

5.5 산술 논리 명령어

```
    segment   .data
word1a      dw      bc62h           ; 데이터 항목
word1b      dw      0123h
word2a      dw      553ah
word2b      dw      0012h
word3a      dw      0                       ; 결과를 저장할 곳
word3b      dw      0                       ;
    segment   .code
    mov       ax, data
    mov       ds, ax
    mov       ax, [word1a]          ; 가장 왼쪽의 워드를 더함
    add       ax, [word2a]
    mov       [word3a], ax           ; 가장 오른쪽의 워드를 캐리와 함께
    mov       ax, [word1b]           ; 더함
    adc       ax, [word2b]
    mov       [word3b], ax
```

첫 번째 MOV와 ADD 연산은 AX 레지스터에서 각 더블워드의 맨 왼쪽워드, 즉 하위 부분을 더한다. 기억할 것은 메모리에는 역순으로 저장된다는 것이다.

5.5 산술 논리 명령어

```
WORD1A:          BC62H
WORD2A:        + 553AH
합               (1)119CH (9C11H가 WORD3A에 저장)
```

- 이 덧셈의 결과는 AX를 초과하기 때문에 캐리가 발생하여 캐리 플래그가 1로 설정된다. 그 다음 덧셈을 수행할 때 만약 ADD를 사용한다면 발생한 캐리를 무시하기 때문에 틀린 결과를 산출한다. 그래서 캐리의 영향을 연산에 포함하는 ADC 명령을 사용하여 덧셈을 수행한다. ADC는 두 수를 더하고 캐리 플래그가 설정되었으므로 더한 합에 1을 더한다.

5.5 산술 논리 명령어

```
WORD1B:          0123H
WORD2B:        + 0012H
Carry          + 1H
-------------------------
합               0136H (0136H가 WORD3B에 저장)
```

- 위 연산을 디버거를 이용하여 추적해보면 AX 레지스터에 0136H가 있고, WORD3A에 순서가 바뀐 9C11H, WORD3B에 역시 순서가 바뀐 3601H가 있는 것을 알 수 있다.

5.5 산술 논리 명령어

```
segment     code
        mov    ax, data
        mov    ds, ax
        clc                                  ; 캐리 플래그를 지움
        mov    cx, 02                        ; 반복 횟수 설정
        mov    si, word1a                    ; 워드 쌍의 맨 왼쪽 워드
        mov    di, word2a                    ; 워드 쌍의 맨 왼쪽 워드
        mov    bx, word3a                    ; 결과 합의 맨 왼쪽 워드
A10:    mov    ax, [si]                      ; 워드를 AX로 이동
        adc    ax, [di]                      ; 캐리와 함께 AX에 더함
        mov    [bx], ax                      ; 결과 합 저장
        inc    si                            ; 오른쪽에 있는 워드를 위해
        inc    si                  ; 주소 조정
        inc    di
        inc    di
        inc    bx
        inc    bx
        loop   a10                 ; 다음 워드로 반복
```

5.5 산술 논리 명령어

- 이 예제는 WORD1a. WORD2a, WORD3a의 주소에 대한 인덱스 레지스터로 각각 SI, DI, BX를 사용한다.
- 더해지는 각 워드 쌍에 대해 명령어들을 두 번씩 반복한다.
- 첫 번째 루프에서 맨 왼쪽 워드들을 더하고, 두 번째 루프에서 맨 오른쪽 워드들을 더한다.
- 두 번째 루프가 워드들을 오른쪽으로 처리하기 때문에 SI, DI, BX 레지스터의 주소는 2씩 증가한다.
- 레지스터의 주소를 증가시킬 때 ADD 대신 INC를 사용한 이유는 캐리 플래그에 영향을 주지 않기 때문이다.
- 프로그램을 시작할 때 CLC(Clear Carry) 명령은 사용한 것은 캐리 플래그를 초기에 지우기 위해서이다. 메모리와 AX 레지스터에 있는 결과는 앞과 동일하다.
- 이 방법을 사용하기 위해서는 다음 조건을 만족해야 한다.
- 워드들을 서로 인접하여 정의하고, CX 레지스터를 더해질 워드의 개수로 초기화하고, 워드를 왼쪽에서 오른쪽으로 처리한다.

5.5 산술 논리 명령어

- SUB 명령어

[라벨:]	SUB	레지스터/메모리, 레지스터/메모리/즉시 값

  - SUB 명령어는 첫 번째 피연산자의 값을 두 번째 피연산자의 값과 뺀 후 결과 값을 첫 번째 피연산자에 저장한다.
  - 첫 번째 피연산자는 레지스터나 메모리가 될 수 있고, 두 번째 피연산자는 레지스터, 메모리, 그리고 즉시 값이 될 수도 있다.
  - SUB 명령어에서 반드시 지켜야할 것은 두 피연산자 모두 메모리가 될 수 없다는 것과 두 피연산자의 크기는 같아야한다는 것이다.

5.5 산술 논리 명령어

- SBB 명령어

[라벨:]	SBB	레지스터/메모리, 레지스터/메모리/즉시 값

  - 이 명령어는 더블워드 이상 수의 뺄셈에 사용된다.
  - 더블워드 크기의 두 수를 워드로 나누어서 뺄셈을 할 때 첫 번째 뺄셈에서 빌림 수가 발생하면, 즉 캐리 플래그가 1로 설정되면 두 번째 뺄셈에 이 빌림을 포함하여 연산을 하는 것이다.
  - SBB 명령어의 첫 번째 피연산자는 레지스터나 메모리가 올 수 있고, 두 번째 피연산자는 레지스터, 메모리, 그리고 즉시 값이 올 수 있다. SBB 명령어도 SUB와 마찬가지로 두 피연산자가 모두 메모리가 될 수 없고, 두 피연산자의 크기는 항상 동일해야 한다.

5.5 산술 논리 명령어

```
                segment  .data
dblword1    dd          762562fah                ; 데이터 정의
dblword2    dd          3412963bh
result      dd          0

                segment .code
    mov         ax, data
    mov         ds, ax
    clc
    mov         ax, word [dblword1]              ; 62FA를 AX에 저장
    sub         ax, word [dblword2]              ; AX - 963B
    mov         word [result], ax                ; 결과 저장
    mov         ax, word [dblword1+2]            ; 7625를 AX에 저장
    sbb         ax, word [dblword2+2]            ; AX - 3412 - 1
    mov         word [result+2], ax              ; 결과 저장
```

5.5 산술 논리 명령어

- 이 예제에서 첫 번째 MOV와 SUB 명령은 두 더블워드의 왼쪽부터 처리하기 위하여 워드 단위로 데이터를 가져와서 뺄셈 연산을 수행한다. 그리고 빌림 수가 있어 캐리 플래그가 1로 설정된다.
- 두 번째 MOV 명령은 결과인 0CCBFH를 메모리에 저장한다.
- 세 번째 MOV 명령은 그 다음 뺄셈을 위해 주소를 2 증가시켜서 워드를 AX 레지스터로 가져온다.
- 앞의 뺄셈에서 캐리 플래그가 설정되어 SBB 명령을 사용하여 빌림수를 반영하여 아래와 같이 뺄셈을 수행한다.
- AX = 7625 – 3412 – 1 = 4212
- 연산 수행 후 AX 레지스터는 4212H가 있고, RESULT에는 4212CCBFH가 있게 된다.

5.5 산술 논리 명령어

- INC/DEC 명령어

[라벨:]	INC/DEC	레지스터/메모리

  - INC와 DEC는 레지스터나 메모리 위치의 내용을 1만큼 증가시키거나 감소시키는데 사용되는 명령어이다.
  - INC와 DEC는 단 1개의 피연산자를 요구한다.
  - 이 연산은 수행 결과에 따라서 OF(부호 비트로 캐리 발생 여부), SF(양수/음수), ZF(0/0이 아님)의 플래그를 지우거나(clear) 설정(set)한다. 조건부 분기 명령어들이 이러한 조건들을 검사할 수 있다.
  - 예를 들어 1-바이트의 값이 FFH를 포함한 경우, INC는 이 값을 00H로 증가시키고 SF를 양수, ZF를 0으로 각각 설정한다. 이 상태에서 DEC는 00H를 FFH로 감소시키고 SF를 음수, ZF를 0이 아님으로 각각 설정한다.
  - ADD와 SUB는 캐리 플래그를 변경시키지만, INC와 DEC는 캐리 플래그를 변경시키지 않는다.

5.5 산술 논리 명령어

- 논리연산 명령어

[라벨:]	AND	레지스터/메모리, 레지스터/메모리/즉시 값
[라벨:]	OR	레지스터/메모리, 레지스터/메모리/즉시 값
[라벨:]	XOR	레지스터/메모리, 레지스터/메모리/즉시 값
[라벨:]	NOT	레지스터/메모리/즉시 값

5.5 산술 논리 명령어

- CMP 명령어

[라벨:]	CMP	레지스터/메모리, 레지스터/메모리/즉시 값

  - 비교 명령은 한 피연산자의 내용을 다른 피연산자의 내용과 비교하기 위해서 사용한다.
  - 첫 번째 피연산자는 레지스터나 메모리이다. 두 번째 피연산자는 레지스터, 메모리, 그리고 즉시 값일 수도 있다.
  - 피연산자들의 조합에서 레지스터의 내용을 메모리나 다른 레지스터의 내용 또는 즉시 값과 비교할 수 있다. 그러나 메모리의 내용은 레지스터의 내용이나 즉시 값과 비교할 수 있으나 다른 메모리 내용과는 비교할 수 없다.

5.5 산술 논리 명령어

```
cmp     ax, bx              ; 레지스터 간의 비교
cmp     cx, [memory]        ; 레지스터와 메모리 비교
cmp     [memory], cx        ; 메모리와 레지스터 비교
cmp     si, 0               ; 레지스터와 즉시 값(수치) 비교
cmp     byte [memory], 'a'      ; 메모리와 즉시 값(문자) 비교
cmp     dl, 41h             ; 레지스터와 즉시 값 비교
```

- 위에 나타낸 예 가운데 마지막 행의 41H는 영문자 'A' 또는 10진수 65가 될 수 있다. 따라서 이 문장이 두 값 중에 어떤 값을 선택할 것인가는 앞뒤의 문장에 따라 결정된다. 이런 경우에는 다음과 같이 사용하는 것이 더 편리하다.

```
    cmp    dl,  'A'          ; DL의 값이 A인가?
    cmp    dl,  65           ; DL의 값이 65인가?
```

5.5 산술 논리 명령어

- 다음에 보인 예는 잘못된 경우이다.
- cmp [xxx], [yyy] ; 메모리 간의 비교
- cmp 'A', ah ; 첫 번째 피연산자가 즉시 값
- cmp ax, bl ; 피연산자의 크기가 다름
- CMP 명령이 두 피연산자의 값을 비교하는 방법은 첫 번째 피연산자의 값에서 두 번째 피연산자의 값을 뺀다. 그리고 뺄셈의 결과에 따라 플래그 레지스터의 해당 플래그를 설정한다. CMP 연산 결과에 의해 영향을 받는 플래그는 AF, ZF, CF, SF, PF, 그리고 OF 이다.
- ZF는 비교 연산의 결과가 '0'일 때 1로 설정되고 나머지 경우에는 0으로 둔다. CF는 비교 연산을 수행할 때 빌림 수가 발생하면 1로 설정되고 그렇지 않으면 0으로 된다. SF는 대상 피연산자가 음수이면 1로 설정되고 양수이면 0이 된다. OF는 부호 있는 산술연산이 대상 피연산자의 크기보다 더 큰 수를 생성할 때 1이 되고 그렇지 않으면 0이 된다. 이러한 플래그는 개별적으로 테스트할 필요는 없다.

5.5 산술 논리 명령어

- 다음 예는 DX 레지스터가 0의 값을 갖는지를 테스트한다.

```
    cmp     dx, 00          ; dx = 0 ?
    je      l10             ; 0이면 l10으로 점프
            ...             ; 0이 아니면 취할 행동
l10:        ...             ; 0인 경우 점프할 지점
```

- DX가 0이면 CMP는 ZF를 1로 설정하고, 다른 플래그들의 설정은 변경하거나 그렇지 않을 수 있다. JE(Jump if Equal) 명령어는 ZF만을 테스트한다. ZF가 1을 포함하기 때문에, JE는 피연산자 l10이 가리키는 주소로 제어를 전달한다.

5.5 산술 논리 명령어

- TEST 명령어
  - TEST 명령은 AND 연산을 수행하지만 결과는 보관하지 않는다.
  - 오직 연산 결과에 따라 플래그 레지스터의 값만을 바꿀 뿐이다.
  - CMP 명령이 뺄셈을 수행하지만 오직 플래그 레지스터의 값만을 바꾸었던 것과 일맥상통한다.
  - 예를 들어, 어떤 연산의 결과가 0이라면 ZF가 세트 된다.

5.6 이동, 회전 명령어

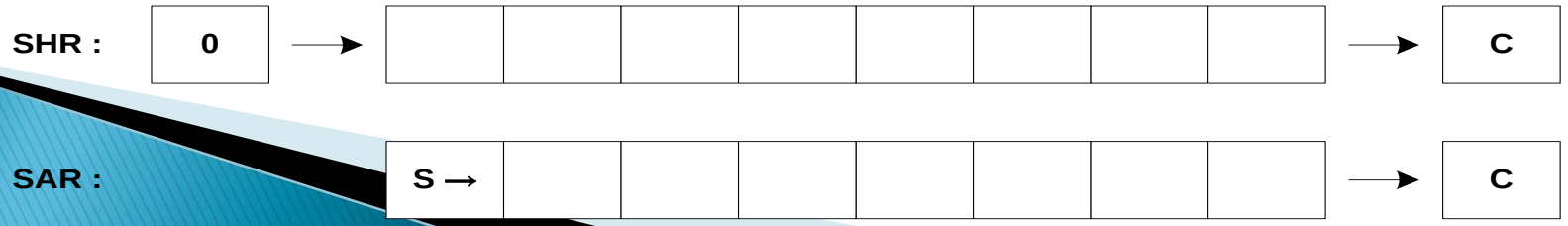
- 비트 이동 명령어(shift instruction)와 비트 회전 명령어(bit rotate instruction)에 대해서 다룬다.
- 비트 이동 명령어는 컴퓨터의 논리 능력의 일부이며 다음 사항을 수행할 수 있다.
- 레지스터나 메모리 주소를 참조, 비트들을 왼쪽이나 오른쪽으로 이동, 바이트에 대해서는 8비트까지, 워드에 대해서는 16비트까지, 더블워드에 대해서는 32비트까지 이동, 논리적(부호가 없는 경우)으로나 산술적(부호가 있는 경우)으로 비트들을 이동시킨다.
- 비트 회전 명령어도 컴퓨터 논리 능력의 일부이며 다음 사항을 수행할 수 있다.
- 레지스터나 메모리 주소를 참조, 비트들을 왼쪽이나 오른쪽으로 회전시킨다. 이동되어 벗어나는 비트는 회전하여 레지스터나 메모리 위치의 비워진 비트에 채워지고, 또한 캐리 플래그에 복사, 바이트에 대해서는 8비트까지, 워드에 대해서는 16비트까지, 더블워드에 대해서는 32비트까지 회전, 논리적(부호가 없는 경우)으로나 산술적(부호가 있는 경우)으로 비트들을 이동시킨다.

5.6 이동, 회전 명령어

- SHR/SAR 명령어

[라벨:]	SHR/SAR	레지스터/메모리, CL/즉시 값

  - SHR, SAR연산은 지정된 레지스터나 메모리 위치에 속한 비트들을 오른쪽으로 이동시킨다.
  - 첫 번째 피연산자는 레지스터나 메모리이고, 두 번째 피연산자는 즉시 값이거나 CL 레지스터의 참조인 비트 이동 값을 포함한다. 8086/8088 프로세서에서 직접 비트 이동 값은 단지 1일 수 있고, 더 큰 값은 반드시 CL 레지스터에 포함되어야 한다. 그 이후의 프로세서에서는 31까지의 직접 비트 이동 값을 허용한다.
  - 이동되어 벗어나는 각 비트는 캐리 플래그로 들어간다. SHR은 논리(무부호) 데이터에 대해서 사용되고, SAR은 산술(부호) 데이터에 대해서 사용된다.



5.6 이동, 회전 명령어

- mov bh, 10110111b ; bh를 초기화
- shr bh, 01 ; 오른쪽으로 1비트 이동
- mov cl, 02 ; 비트 이동 값 설정
- shr bh, cl ; 오른쪽으로 2비트 이동
- shr bh, 02 ; 오른쪽으로 2비트 이동
- 첫 번째 SHR은 BH 레지스터의 내용을 1비트 오른쪽으로 이동, 이동되어 벗어난 1-비트는 캐리 플래그에 적재, 0이 BH의 왼쪽에 채워진다. 두 번째 SHR은 BH를 2비트 더 이동, 캐리 플래그는 순차적으로 1, 1을 포함하고 두 개의 0이 BH의 왼쪽에 채움. 세 번째 SHR은 BH를 2비트 더 이동시킨다.

명령어	BH	10진수	CF
mov	10110111	183	
shr	01011011	91	1
shr	00010110	22	1
shr	00000101	5	1

5.6 이동, 회전 명령어

- SAR은 SHR과 다른 한 가지 중요한 차이가 있다. 그 차이점은 비트 이동 후 맨 왼쪽의 비워진 비트를 부호 비트(sign bit)를 이용해서 채우는 것이다. 이러한 방식으로 양수와 음수는 그 부호를 유지한다.

```
mov     bh, 10110111b ; bh를 초기화
sar     bh, 01        ; 오른쪽으로 1비트 이동
mov     cl, 02        ; 비트 이동 값 설정
sar     bh, cl        ; 오른쪽으로 2비트 이동
sar     bh, 02        ; 오른쪽으로 2비트 이동
```

명령어	BH	10진수	CF
mov	10110111	-73	
sar	11011011	-36	1
sar	11110110	-9	1
sar	11111101	-2	1

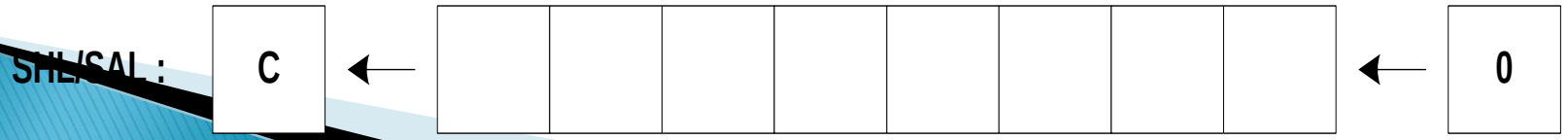
- 오른쪽 방향 비트 이동을 살펴보면, 값을 절반으로 나누는 것을 알 수 있다. 5나 7과 같은 홀수는 2로 나누면 각각 2와 3이 생성되고 캐리 플래그가 1로 설정된다. 비트 이동 연산 후 사용자는 JC(Jump If Carry)를 사용하여 캐리 플래그를 테스트할 수 있다.

5.6 이동, 회전 명령어

- SHL/SAL 명령어

[라벨:]	SHL/SAL	레지스터/메모리, CL/즉시 값

  - SHL, SAL연산은 지정된 레지스터나 메모리 위치에 속한 비트들을 왼쪽으로 이동시킨다.
  - 첫 번째 피연산자는 레지스터나 메모리이고, 두 번째 피연산자는 즉시 값이거나 CL 레지스터의 참조인 비트 이동 값을 포함한다. 8086/8088 프로세서에서 직접 비트 이동 값은 단지 1일 수 있고, 더 큰 값은 반드시 CL 레지스터에 포함되어야 한다. 그 이후의 프로세서에서는 31까지의 직접 비트 이동 값을 허용한다.
  - 이동되어 벗어나는 각 비트는 캐리 플래그로 들어간다. SHL은 논리(무부호) 데이터에 대해서 사용되고, SAL은 산술(부호) 데이터에 대해서 사용된다.



5.6 이동, 회전 명령어

- mov bh, 00000101b ; bh를 초기화
- shl bh, 01 ; 왼쪽으로 1비트 이동
- mov cl, 02 ; 비트 이동 값 설정
- shl bh, cl ; 왼쪽으로 2비트 더 이동
- shl bh, 02 ; 왼쪽으로 2비트 더 이동
- 첫 번째 SHL은 BH의 내용을 한 비트 왼쪽으로 이동시킨다. 이동되어 벗어난 0-비트는 캐리 플래그에 적재되고, 0이 BH 레지스터의 오른쪽에 채워진다. 두 번째 SHL은 BH의 내용을 두 개 비트 더 이동시킨다. 따라서 캐리 플래그는 순차적으로 0, 0을 포함하고, 두 개의 0이 오른쪽에 채워진다. 세 번째 SHL은 BH를 두 개 비트 더 이동시킨다.

5.6 이동, 회전 명령어

명령어	BH	10진수	CF
mov	00000101	5	
shl	00001010	10	0
shl	00101000	40	0
shl	10100000	160	0

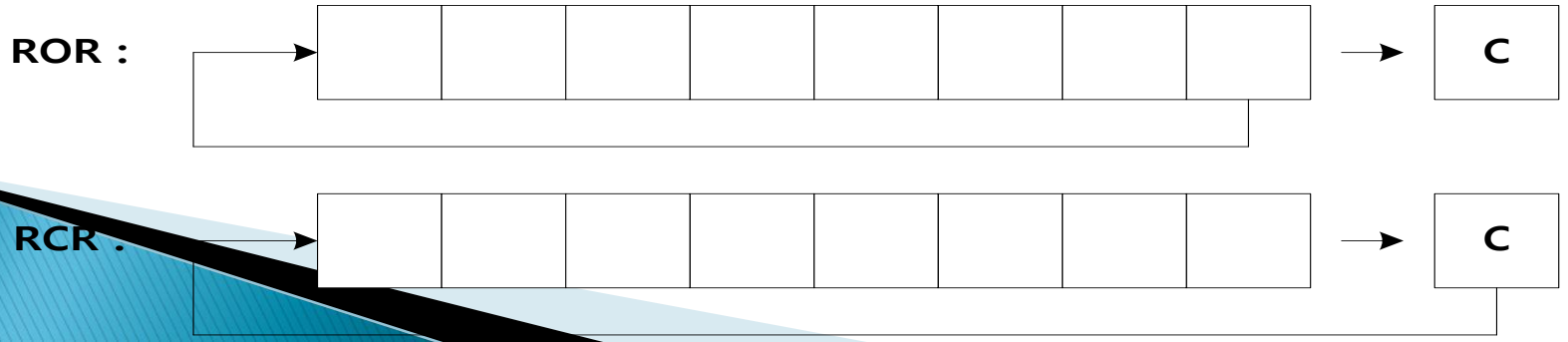
- 왼쪽 방향 비트 이동은 항상 오른쪽을 0으로 채운다. 그 결과 SHL과 SAL은 동일하다. 따라서 위 예제에서 SAL을 사용하여도 똑같은 결과가 나온다.
- 왼쪽 방향 비트 이동의 결과를 살펴보면 한 개 비트 이동할 때마다 값을 두 배가 된다는 것을 알 수 있다. 그래서 왼쪽 방향 비트 이동은 값을 두 배로 하는데 특히 유용하며, 곱셈 연산보다 훨씬 더 빠르게 수행된다. 위 예제의 첫 번째 SHL은 실제로 2를 곱하고, 두 번째와 세 번째 SHL은 각각 4를 곱한다.
- 비트 이동 연산 후 사용자는 JC 명령어를 사용하여 캐리 플래그로 이동된 비트들을 테스트할 수 있다.

5.6 이동, 회전 명령어

- ROR/RCR 명령어

[라벨:]	ROR/RCR	레지스터/메모리, CL/즉시 값

  - ROR, RCR연산은 지정된 레지스터나 메모리 위치에 속한 비트들을 오른쪽으로 회전시킨다.
  - 첫 번째 피연산자는 레지스터나 메모리이고, 두 번째 피연산자는 즉시 값이거나 CL 레지스터의 참조인 비트 이동 값을 포함한다. 8086/8088 프로세서에서 직접 비트 이동 값은 단지 1일 수 있고, 더 큰 값은 반드시 CL 레지스터에 포함되어야 한다. 그 이후의 프로세서에서는 31까지의 직접 비트 이동 값을 허용한다.
  - 이동되어 벗어나는 각 비트는 캐리 플래그로 들어간다. ROR은 논리(무부호) 데이터에 대해서 사용되고, RCR은 산술(부호) 데이터에 대해서 사용된다.



5.6 이동, 회전 명령어

```
mov    bl, 10110111b        ; bh를 초기화
ror    bl, 01               ; 오른쪽으로 1비트 회전
mov    cl, 03               ; 비트 회전 값 설정
ror    bl, cl               ; 오른쪽으로 3비트 더 회전
ror    bl, 03               ; 오른쪽으로 3비트 더 회전
```

- 첫 번째 ROR은 BL 레지스터의 맨 오른쪽 1을 회전시켜 맨 왼쪽의 비워진 곳과 CF로 이동시킨다. 두 번째와 세 번째 ROR은 맨 오른쪽으로부터 세 개의 비트를 회전시켜 왼쪽의 비워진 곳과 CF도 이동시킨다.

명령어	BL	CF
mov	10110111	
ror	11011011	1
ror	01111011	0
ror	01101111	0

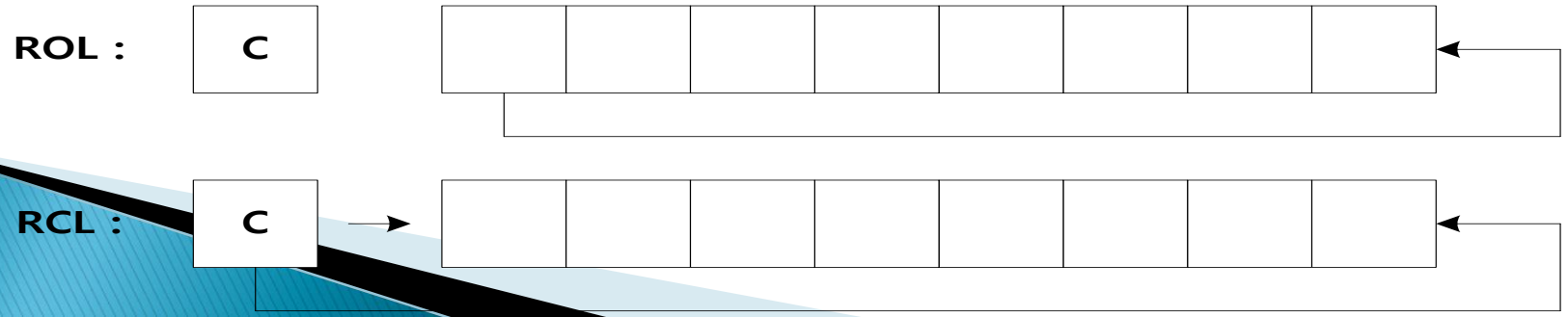
- RCR은 ROR과 동작 진행에 있어서 한 가지 다른 점이 있다. RCR은 오른쪽으로 벗어나는 각 비트를 먼저 CF도 이동하고, CF에 적재된 이 비트를 레지스터나 메모리의 비워진 비트 위치로 이동, RCR 연산 후 사용자는 JC 명령어를 사용하여 CF로 회전 이동된 비트를 테스트할 수 있다.

5.6 이동, 회전 명령어

- ROL/RCL 명령어

[라벨:]	ROL/RCL	레지스터/메모리, CL/즉시 값

  - ROL, RCL연산은 지정된 레지스터나 메모리 위치에 속한 비트들을 왼쪽으로 회전시킨다.
  - 첫 번째 피연산자는 레지스터나 메모리이고, 두 번째 피연산자는 즉시 값이거나 CL 레지스터의 참조인 비트 이동 값을 포함한다. 8086/8088 프로세서에서 직접 비트 이동 값은 단지 1일 수 있고, 더 큰 값은 반드시 CL 레지스터에 포함되어야 한다. 그 이후의 프로세서에서는 31까지의 직접 비트 이동 값을 허용한다.
  - 이동되어 벗어나는 각 비트는 캐리 플래그로 들어간다. ROL은 논리(무부호) 데이터에 대해서 사용되고, RCL은 산술(부호) 데이터에 대해서 사용된다.



5.6 이동, 회전 명령어

- mov bl, 10110111b ; bh를 초기화
- rol bl, 01 ; 왼쪽으로 1비트 회전
- mov cl, 03 ; 비트 이동 값 설정
- rol bl, cl ; 왼쪽으로 3비트 더 회전
- rol bl, 03 ; 왼쪽으로 3비트 더 회전
- 첫 번째 ROL은 BL 레지스터의 맨 왼쪽 1을 회전시켜 맨 오른쪽의 비워진 곳과 CF로 이동시킨다. 두 번째와 세 번째 ROL은 맨 왼쪽부터 세 개 비트를 회전시켜 오른쪽의 비워진 곳과 CF로 순차적으로 이동시킨다.

명령어	BH	CF
mov	10110111	
rol	01101111	1
rol	01111011	1
rol	11011011	1

- RCL은 ROL과 동작 진행에 있어서 한 가지 다른 점이 있다. RCL은 왼쪽으로 벗어나는 각 비트를 먼저 CF도 이동하고, CF에 적재된 이 비트를 레지스터나 메모리의 비워진 비트 위치로 이동, RCL 연산 후 사용자는 JC 명령어를 사용하여 CF로 회전 이동한 비트를 테스트할 수 있다.

5.7 곱셈, 나눗셈 명령어

- 곱셈 명령어

[라벨:]	MUL/IMUL	레지스터/메모리

  - 2진수 데이터의 곱셈에서 데이터가 부호가 없으면 MUL(Multiply) 명령어를 사용하고, 2의 보수이면 IMUL(Integer Multiply) 명령어를 사용한다. 두 명령어는 캐리 플래그와 오버플로 플래그에 영향을 미친다.
  - 피연산자는 8비트, 16비트 및 32 비트 레지스터나 기억장소이며, 즉시 값이나 상수 값은 피연산자로 사용될 수 없다.
  - 곱셈 연산은 바이트×바이트, 워드×워드, 더블워드×더블워드가 있다.

5.7 곱셈, 나눗셈 명령어

- 바이트 곱셈
  - 두 개의 1 바이트 값들을 곱셈하는 경우, 피승수는 AL 레지스터에 있으며, 승수는 메모리 또는 다른 레지스터에 있는 1 바이트이다.
  - 예를 들어 MUL DL 명령어에서 연산은 AL 레지스터의 값과 DL 레지스터의 값을 곱한다.
  - 생성된 곱셈의 결과는 AX 레지스터에 저장된다. 이 연산은 곱셈 연산 전에 이미 들어 있는 AH 레지스터의 값은 무시하고 지운다. 곱셈 전, 후의 AX 레지스터의 상태를 아래 그림에 보였다.

5.7 곱셈, 나눗셈 명령어

- 워드 곱셈
  - 두 개의 1 워드 값들을 곱하는 경우, 피승수는 AX 레지스터에 있으며, 승수는 메모리 또는 다른 레지스터에 있는 1 워드이다.
  - 예를 들어 MUL DX 명령어에서 연산은 AX 레지스터의 값과 DX 레지스터의 값을 곱한다.
  - 곱셈의 결과로 생성된 값은 더블워드이다. 따라서 저장하기 위해서는 두 개의 레지스터가 필요하다. 결과의 상위 부분은 DX에 저장되고, 하위 부분은 AX에 저장된다. 이 연산은 DX에 이미 저장되어 있는 데이터를 무시하고 지운다. 아래에 곱셈 전, 후의 레지스터 상태를 그림으로 보였다.

	DX	AX
곱셈 전:	(무시됨)	피승수
곱셈 후:	곱의 상위	곱의 하위

5.7 곱셈, 나눗셈 명령어

- 2중 워드 곱셈
  - 두 개의 더블워드를 곱하는 경우, 피승수는 EAX 레지스터에 있고, 승수는 메모리 또는 다른 레지스터에 있는 하나의 더블워드이다.
  - 곱셈의 결과는 EDX:EAX 레지스터 쌍에 생성된다.
  - 이 연산은 이미 EDX 레지스터에 들어있는 데이터를 무시하고 지운다. 아래에 곱셈 전, 후의 레지스터 상태를 그림으로 보였다.

	EDX	EAX
곱셈 전:	(무시됨)	피승수
곱셈 후:	곱의 상위	곱의 하위

5.7 곱셈, 나눗셈 명령어

- 필드 크기
  - MUL과 IMUL 연산의 피연산자는 승수만 참조하는데 이것이 필드의 크기를 결정한다. 이 명령어는 피승수가 승수의 크기에 따라 AL, AX, EAX 레지스터에 있다고 가정한다. 다음에 나타낸 예제에서 승수는 바이트, 워드, 더블워드로서 레지스터에 있다.

명령어	승수	피승수	곱
mul cl	바이트	al	ax
mul bx	워드	ax	ax:dx
mul ebx	더블워드	eax	eax:edx

```
byte1   resb   1   ; 바이트 값
word1   resw   1   ; 워드 값
dword1  resd   1   ; 더블워드 값
```

명령어	승수	피승수	곱
mul byte [byte1]	byte1	al	ax
mul word [word1]	word1	ax	ax:dx
mul dword [dword1]	dword1	eax	eax:edx

5.7 곱셈, 나눗셈 명령어

- 나눗셈 명령어

[라벨:]	DIV/IDIV	레지스터/메모리

  - 나눗셈에서 DIV(Divide) 명령어는 부호 없는 데이터에 사용되고, IDIV(Integer Divide) 명령어는 부호 있는 데이터에 사용된다.
  - 피연산자는 레지스터나 메모리이고 즉시 값이나 상수 값은 피연산자로 사용될 수 없다. 기본적인 나눗셈 연산은 워드/바이트, 더블워드/워드, 그리고 쿼드워드/더블워드이다.

5.7 곱셈, 나눗셈 명령어

- 워드 나누기 바이트
  - 피제수는 AX 레지스터에 있고, 제수는 메모리 또는 다른 레지스터에 있는 1 바이트 데이터이다.
  - 이 연산은 나눗셈 후 나머지는 AH 레지스터에 저장하고 몫은 AL 레지스터에 저장한다.
  - 1 바이트 몫은 부호 없는 경우 최대 +255(FFH), 부호 있는 경우 최대 +127(7FH)까지로 매우 작다는 것에 유의해야 한다. 나눗셈 수행 전, 후의 레지스터의 상태를 아래 그림에 나타내었다.

5.7 곱셈, 나눗셈 명령어

- 2중 워드 나누기 워드
  - 더블워드를 워드로 나누는 경우에 피제수는 DX:AX 레지스터 쌍에 있고, 제수는 메모리 또는 다른 레지스터에 있는 워드이다.
  - 이 연산은 나눈 후 나머지를 DX 레지스터에 저장하고 몫을 AX 레지스터에 저장한다.
  - 한 워드의 몫은 부호 없는 경우 최대 +65,535(FFFFH)이고, 부호 있는 경우 최대 +32,767(7FFFH)이다. 나눗셈 전, 후의 레지스터 상태를 다음 그림에 나타내었다.

	DX	AX
나눗셈 전:	상위 피제수	하위 피제수
나눗셈 후:	나머지	몫

5.7 곱셈, 나눗셈 명령어

- 4중 워드 나누기 2중 워드
  - 쿼드워드를 더블워드로 나누는 경우에 피제수는 EDX:EAX 쌍에 있고, 제수는 메모리 또는 다른 레지스터에 있는 더블워드이다.
  - 이 연산은 나눈 후 나머지를 EDX 레지스터에 저장하고, 몫을 EAX 레지스터에 저장한다. 나눗셈을 수행하기 전, 후의 레지스터 상태를 다음 그림에 나타내었다.

	EDX	EAX
나눗셈 전:	상위 피제수	하위 피제수
나눗셈 후:	나머지	몫

5.7 곱셈, 나눗셈 명령어

- 필드 크기
  - DIV/IDIV의 피연산자는 제수를 참조하며 제수가 필드의 크기를 결정한다.

명령어	제수	피제수	몫	나머지
div cl	바이트	ax	Al	Ah
div cx	워드	dx:ax	ax	dx
div ebx	더블워드	edx:eax	eax	edx

```
byte1    resb    1      ; 바이트 값
word1    resw    1      ; 워드 값
dword1   resd    1      ; 더블워드 값
```

명령어	제수	피제수	몫	나머지
div byte [byte1]	byte1	ax	al	ah
div word [word1]	word1	dx:ax	ax	dx
div dword [dword1]	dword1	edx:eax	eax	edx

5.8 제어 명령어

- 현재 실행되고 있는 명령어 바로 다음에 위치한 명령어를 실행할 것인지 아니면 떨어져 있는 명령어를 실행할 것인지를 선택하는 제어를 전달하는 명령어들이다.
- 제어의 전달 방향은 새로운 단계를 실행하기 위해서 정방향(forward)일 수도 있고, 동일한 단계를 다시 실행하기 위해서 역방향(backward)일 수도 있다.
- 조건 없이 분기하는 무조건분기 명령에 대해서 설명한 후에, 조건에 따라 분기하는 조건 분기 명령을 살펴본다. 또한, 명령들을 반복 실행하는 반복 명령을 다룬다.

5.8 제어 명령어

- JMP 명령어

[라벨:]	JMP	단거리/근거리/원거리 주소

  - JMP 명령어는 프로그램을 수행하다가 조건 없이 분기하도록 하는 명령이다.
  - 제어를 전달하는 데 일반적으로 사용되는 명령어이다. JMP는 프로세서의 사전반입(prefetch) 명령어 큐를 비운다. 따라서 많은 점프 연산을 갖는 프로그램은 처리속도가 떨어질 수 있다.

5.8 제어 명령어

- 단거리와 근거리 점프
  - 어셈블러는 현재의 주소로부터 떨어진 정도에 따라 구분되는 세 가지 유형의 주소를 지원한다.
  - 단거리(short) 주소: -128(80H) ~ 127(7FH)바이트까지의 거리로 제한.
  - 근거리(near) 주소: 동일 세그먼트 내에서 -32,768(8000H) ~ 32,767(7FFFH) 바이트까지의 거리로 제한.
  - 원거리(far) 주소: 코드 세그먼트(CS)가 다른 주소.
  - JMP 연산은 같은 세그먼트 내에서 단거리이거나 근거리 점프일 수 있다.
  - -128(80H) ~ 127(7FH) 바이트의 범위에 속한 라벨에 대한 JMP 연산은 단거리 점프(short jump)이다. 피연산자는 프로그램이 실행될 때 프로세서가 IP(Instruction Pointer) 레지스터에 더하는 오프셋 값으로 사용된다.
  - 단거리 주소를 초과하고 32K 이내의 범위에 속한 라벨에 대한 JMP 연산은 근거리 점프(near jump)이다.
  - 원거리 점프(far jump)는 코드 세그먼트(CS)와 IP가 모두 바뀌는 경우이다.

5.8 제어 명령어

- 역방향과 정방향 점프
  - 점프는 역방향(전방참조)과 정방향(후방참조)일 수 있다. 어셈블러는 다음과 같이 이미 -128바이트 내에 속한 지정된 피연산자[역방향 점프(backward jump)]를 만날 수 있을 것이다.
  - j10: ; 점프 주소
  - ...
  - jmp j10 ; 역방향 점프.
  - 정방향 점프(forward jump)인 경우에 어셈블러는 지정된 피연산자를 아직 만나지 않았다.
  - jmp j20 ; 정방향 점프
  - ...
  - j20: ; 점프 주소.
  - 역방향 점프는 분기할 주소와 현재 위치를 어셈블러가 알 수 때문에 단거리, 근거리 또는 원거리를 명시해주지 않아도 된다. 그러나 정방향 점프는 분기할 주소를 모르기 때문에 어셈블러는 근거리 점프로 가정한다. 프로그래머는 정방향 참조일 때 필요하다면 이를 구분해주어야 한다.

5.8 제어 명령어

- 직접 분기와 간접 분기
  - 'jmp xxx' 명령은 'xxx' 번지로 분기하라는 직접 점프 명령이다. 'jmp [xxx]' 명령은 'xxx' 번지에 기억된 번지로 분기하라는 간접 점프 명령이다.
  - xxx dw yyy
  - aaa resd 1
  - .....
  - jmp [xxx]
  - jmp dword [aaa]
  - jmp far bbb.
  - 'jmp [xxx]'은 xxx 번지에 기억된 번지, 즉 'yyy' 번지로 근거리 점프하라는 명령이다. 즉, 코드 세그먼트는 변경하지 않고, IP만 변경된다.
  - 'jmp dword [aaa]'는 'aaa'에 기억된 번지에서 16비트 IP를, 'aaa+2'에 기억된 번지에서 16비트 코드 세그먼트(CS)를 읽어 와서 그곳으로 원거리 점프하라는 명령이다.
  - 'jmp far bbb' 명령은 bbb 번지로 분기하는데 코드 세그먼트와 IP를 모두 변경하라는 명령이다. 'bbb'가 전방참조, 즉 역방향 점프이면 'far'를 명시해주지 않아도 되지만, 후방참조(정방향 점프)라면 반드시 'far'를 명시해주어야 한다.

    간접 분기의 경우에 'dword'가 없으면 항상 근거리 점프를 한다. 원거리 점프는 전방참조나 후방참조에 상관없이 반드시 'dword'를 명시해주어야 한다.

5.8 제어 명령어

- 프로그램: JMP 명령어 사용
  - segment code
  - ..start:
  - mov ax, 00 ; AX, BX, CX 초기화
  - mov bx, 00 ;
  - mov cx, 01 ;
  - a20: add ax, 01 ;
  - add bx, ax ;
  - shl cx, 01 ;
  - jmp a20 ; 반복
  - 이 프로그램은 AX와 BX 레지스터를 0으로 초기화하고, CX 레지스터를 1로 초기화한다.
  - 루프의 끝에서 명령어 JMP A20은 제어를 A20의 라벨을 갖는 명령어로 전달한다. 루프의 반복 효과로 AX가 1, 2, 3, 4, ... 등으로 증가하게 되고, BX는 1, 3, 6, 10, ... 등으로 숫자들의 합계에 따라 증가하게 되고, CX는 1, 2, 4, 8, ... 등으로 두 배씩 증가하게 된다.
  - 이 루프는 종료되지 않기 때문에 처리는 계속된다 – 보통은 단지 모니터링 시스템과 같은 특정 응용프로그램에서만 사용하는 형태이다.
  - DEBUG를 사용하여 프로그램을 실행하여 반복 횟수를 알아보고, AX, BX, CX, IP 상에서 실행 효과를 살펴보라. 8번의 반복이 이루어진 후에 AX는 08을, BX는 36(24H)을, CX는 256(100H)을 각각 포함한다.

5.8 제어 명령어

- 조건부 점프 명령어
  - 프로세서는 플래그 레지스터의 다양한 플래그들의 설정에 따라 제어를 전달하는 다양한 조건부 점프 명령을 지원한다. 예를 들어, 사용자는 두 개의 필드를 비교하거나 더해서, 각 연산이 설정하는 플래그의 값에 따라서 조건부로 점프할 수 있다.

[라벨:]	Jnnn	단거리 주소

  - 명령어에서 J는 Jump를 의미하고, nnn은 조건을 의미한다. 조건은 부호를 포함하는 것과 포함하지 않는 것으로 나뉜다.
  - 8086/80286의 경우 조건부 점프에 대한 거리는 -128(80H) ~ 127(7FH) 바이트이내의 단거리여야 한다. 80386 이상 프로세서는 32K 내의 임의 주소로 전달하는 것을 가능하게 하는 32-비트(근거리) 오프셋을 제공한다.
  - 이때도 전방참조는 'near'를 명시하지 않아도 되나, 후방참조인 경우에는 반드시 'near'를 명시해주어야 근거리 점프가 가능하다.
    조건부 점프는 원거리 점프와 간접 분기는 불가능하다.

5.8 제어 명령어

- 부호와 무부호 데이터
  - 사용자는 비교나 산술 연산을 수행하고 있는 데이터의 유형(무부호나 부호)에 따라서 사용할 명령이 결정될 수 있다. 무부호(unsigned) 수치 항목은 모든 비트들을 데이터 비트로 다룬다.
  - 대표적 예는 고객 번호, 전화 번호, 비율 같은 수치 값들이다. 부호(signed) 수치 항목은 2의 보수 연산의 결과를 반영한다. 대표적인 예는 수량, 은행 잔고, 온도 등이다.

명령어	설 명	테스트 플래그
JE/JZ	Jump Equal 또는 Jump Zero	ZF
JNE/JNZ	Jump Not Equal 또는 Jump Not Zero	ZF
JB/JNAE	Jump Below 또는 Jump Not Above Equal	CF
JBE/JNA	Jump Below/Equal 또는 Jump Not Above	AF, CF
JA/JNBE	Jump Above 또는 Jump Not Below Equal	CF, ZF
JAE/JNB	Jump Above/Equal 또는 Jump Not Below	CF

5.8 제어 명령어

- 부호(산술) 데이터에 기초한 조건부 점프 명령어를 다음에 나타내었다.
- JE/JZ와 JNE/JNZ를 테스트하는 점프는 무부호와 부호 데이터의 두 리스트 모두에 포함되어 있다. 이것은 이 조건이 부호의 존재 여부에 관계없이 존재하기 때문이다.

명령어	설명	테스트 플래그
JE/JZ	Jump Equal 또는 Jump Zero	ZF
JNE/JNZ	Jump Not Equal 또는 Jump Not Zero	ZF
JL/JNGE	Jump Less 또는 Jump Not Greater/Equal	OF, SF
JLE/JNG	Jump Less/Equal 또는 Jump Not Greater	OF, SF, ZF
JG/JNLE	Jump Greater 또는 Jump Not Less/Equal	OF, SF, ZF
JGE/JNL	Jump Greater/Equal 또는 Jump Not Less	OF, SF

5.8 제어 명령어

- 다음으로 특수한 용도를 갖는 조건부 점프 명령어들은 다음에 나타내었다.
- JCXZ는 CX의 내용이 0인지를 테스트한다. 이 명령어는 산술이나 비교 연산 다음에 올 필요는 없다. JCXZ의 한 가지 용도는 루프의 시작 부분에 위치하여, CX의 초기 값이 0이면 이 루프의 루틴을 실행하지 않고 건너뛰는 것을 보장하는 것이다. JC와 JNC는 흔히 디스크 연산의 성공이나 실패 여부를 테스트하기 위해서 사용된다.
- 무부호 데이터에 대한 점프는 equal, above, 또는 below를 쓰고, 부호 데이터에 대한 점프는 equal, gerater, 또는 less라는 것에 유의하라.

명령어	설명	테스트 플래그
JCXZ	Jump if CX is Zero	없음
JC	Jump Carry(JB와 동일)	CF
JNC	Jump No Carry	CF
JO	Jump Overflow	OF
JNO	Jump No Overflow	OF
JP/JPE	Jump Parity 또는 Jump Parity Even	PF
JNP/JPO	Jump No Parity 또는 Jump Parity Odd	PF
JS	Jump Sign(음수)	SF
JNS	Jump No Sign(양수)	SF

5.8 제어 명령어

- LOOP 명령어
  - 특정 조건에 도달할 때까지만 반복하도록 하는 명령어가 LOOP이다.
  - 이 명령어는 CX 레지스터에 초기 값을 요구한다. 그리고 각 반복에 대하여 LOOP는 CX를 자동으로 1만큼 감소시킨다. CX가 0이 되면 제어는 LOOP의 다음 명령어로 전달되고, 아직 0이 아니면 제어는 LOOP의 피연산자 주소로 점프한다. 피연산자까지의 거리는 단거리 점프이다. 즉, -128(80H) ~ 127(7FH)바이트 이내에 속해야 한다.

[라벨:]	LOOP	단거리 주소

5.8 제어 명령어

- 특정 조건에 도달할 때까지만 반복하도록 하는 명령어가 LOOP이다.
- 이 명령어는 CX 레지스터에 초기 값을 요구한다. 그리고 각 반복에 대하여 LOOP는 CX를 자동으로 1만큼 감소시킨다. CX가 0이 되면 제어는 LOOP의 다음 명령어로 전달되고, 아직 0이 아니면 제어는 LOOP의 피연산자 주소로 점프한다. 피연산자까지의 거리는 단거리 점프이다. 즉, -128(80H) ~ 127(7FH)바이트 이내에 속해야 한다.
- 다음은 프로그램은 LOOP의 사용을 예로 나타낸 것이다. 이 프로그램은 CX 레지스터의 값을 8로 초기화하고, 8번의 반복 후에 종료되는 것을 제외하면 앞의 jmp 프로그램과 같은 연산을 수행한다.

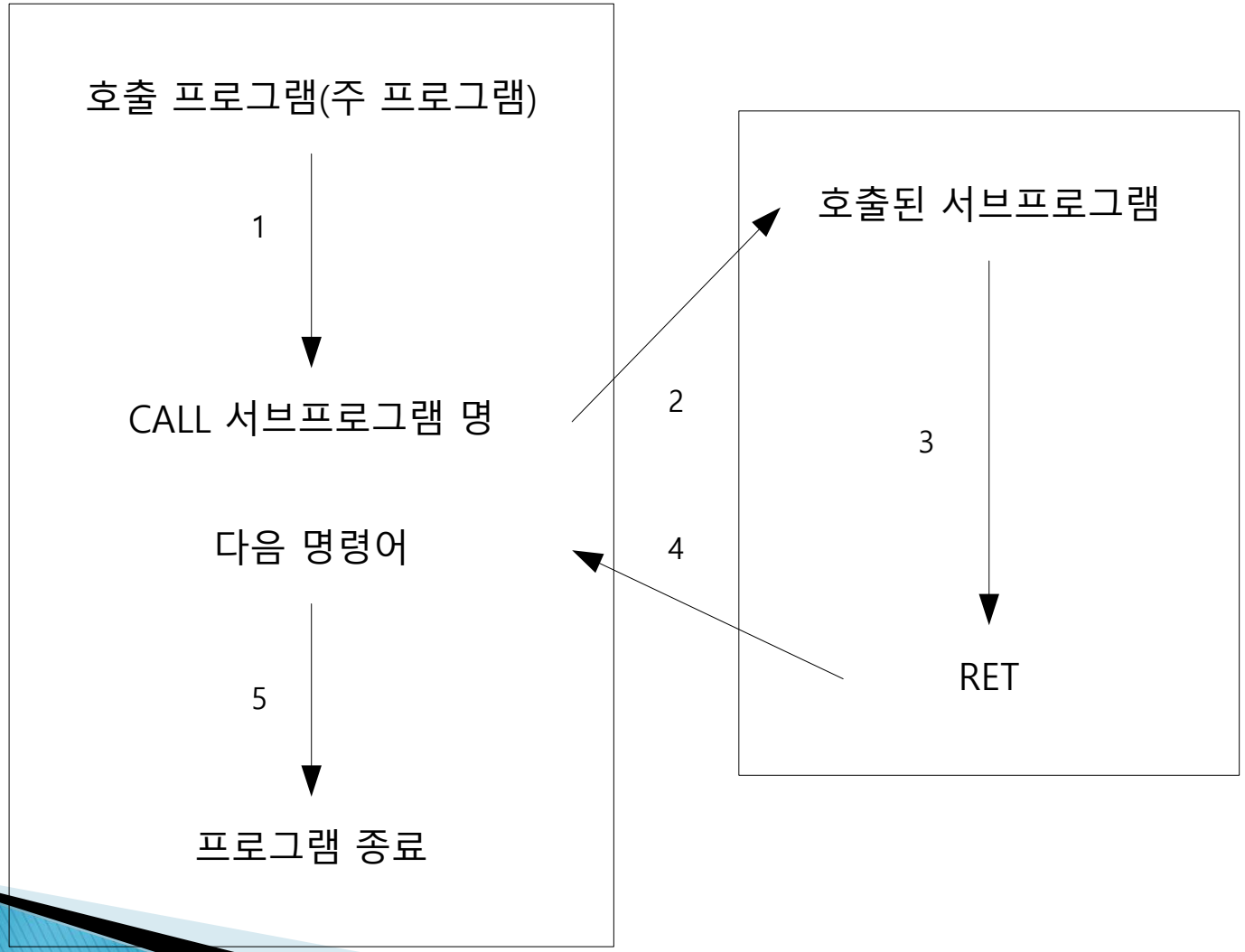
```
segment     code
..start:
            mov     ax, 00          ; AX, BX, DX 초기화
            mov     bx, 00          ;
            mov     dx, 01          ;
            mov     cx, 8               ; 반복 횟수 초기화
A20:        inc     ax                          ;
            add     bx, ax          ;
            shl     dx, 01          ;
            loop    a20                         ; CX를 1씩 감소

            mov     ax, 4c00h       ; 프로세스 종료
            int     21h             ;
```

5.9 서브프로그램 명령어

- 서브프로그램은 호출프로그램이나 다른 서브프로그램에 의해서 사용될 수 있는 독립된 프로그램이다.
- 호출프로그램이 서브프로그램을 사용하는 것을 호출(call)한다고 한다. 서브프로그램은 호출프로그램으로부터 값을 받아서 처리한 후 결과를 호출프로그램으로 반환한다.
- 어떤 프로그램이 프로그램 내의 한 지점에서 서브프로그램을 호출하면 호출된 지점을 기억한 후에 프로그램의 제어를 서브프로그램으로 넘겨준다.
- 서브프로그램 수행 후 기억한 호출된 지점을 복원한 후 호출프로그램으로 돌아온다. 이 방법은 호출프로그램이 마치지 않은 상태에서 서브프로그램이 수행되므로 IP(EIP) 값이 변한다.
- 따라서 서브프로그램 수행 후 호출프로그램으로 돌아오기 위한 복귀 주소를 저장해야 한다. 또한 서브프로그램을 수행하기 전에 사용했던 레지스터의 값들도 저장을 해야 한다.
- 이들을 저장하는데 사용하는 것이 스택(stack)이다.

5.9 서브프로그램 명령어



5.9 서브프로그램 명령어

- CALL과 RET 명령어
  - 스택을 사용해서 서브프로그램을 빠르고, 편하게 호출하기 위해 두 가지 명령을 지원하는데 CALL과 RET 명령이다.

[라벨:]	CALL	서브프로그램 이름
[라벨:]	RET[n]	[직접 값]

  - CALL 명령은 다음에 실행될 명령의 주소를 스택에 저장하고, 서브프로그램의 주소로 분기한다.
  - RET 명령은 스택에 저장된 복귀 주소를 꺼내온 후 그 주소로 점프한다. 즉 피호출 서브프로그램으로부터 호출 프로그램으로 돌아오는 것이다.
  - RET는 피호출 서브프로그램의 끝에 위치한다.

5.9 서브프로그램 명령어

```
        segment     .data

        segment     .stack stack

        segment     .code
        global  _a_main
_a_main:
        call        b_sub
        ...
        mov         ax, 4c00h
        int         21h
b_sub:
        call        c_sub
        ...
        ret
c_sub:
        ...
        ret
```

5.9 서브프로그램 명령어

- 프로그램은 한 개의 주프로그램 _a_main과 두 개의 서브프로그램 b_sub와 c_sub로 구분된다. 각 프로그램은 고유한 이름을 가지며, 끝에는 ret 명령어를 포함한다.주프로그램 _a_main에서 call은 프로그램 제어를 b_sub로 전달하고, b_sub의 실행을 시작한다.
- 서브프로그램 b_sub에서 call은 프로그램 제어를 c_sub로 전달하고, c_sub의 실행을 시작한다.
- 서브프로그램 c_sub에서 ret는 call c_sub 바로 다음에 오는 명령어로 복귀시킨다.
- 서브프로그램 b_sub에서 ret는 call _a_main 바로 다음에 오는 명령어로 복귀시킨다.
- 다음에 주프로그램 _a_main은 복귀 지점에서부터 처리를 재개한다.
- ret는 항상 그 복귀지점으로 복귀한다. 만약 b_sub가 ret로 끝나지 않았으면 처리는 b_sub을 거치고, 직접 c_sub로 연결된다. c_sub가 ret로 끝나지 않았으면 처리는 c_sub의 끝을 지나고, 다음에 어떤 명령어가 오든지 이를 계속 실행한다. 따라서 예측할 수 없는 결과가 초래될 것이다.
- CALL 명령도 근거리(near), 원거리(far), 간접 근거리 CALL 및 간접 원거리 CALL이 있다. 'CALL far xxx'는 'xxx' 번지로 분기하는데 코드 세그먼트와 IP를 모두 변경하라는 명령이다. 다음에 실행할 주소를 스택에 저장할 때 코드 세그먼트, IP 순서로 스택에 저장한다. RET 또한 근거리 RET(RET)과 원거리 RET(RETF)의 두 가지 명령어가 있다.