응용보안

3. 80×86 시스템의 이해 2

경기대학교 AI컴퓨터공학부 이재흥
jhlee@kyonggi.ac.kr

KGU 경기대학교

CONTENTS
PRESENTATION

- 어셈블리어 기본 문법과 명령
- 스택을 이용한 명령 처리 과정

학습 목표

- 어셈블리어를 이해한다.
- 80×86 시스템 스택에서 명령을 처리하는 구조를 이해한다.

어셈블리어 기본 문법과 명령

어셈블리어의 구조

- 어셈블리어의 구조
  - Intel 문법과 AT&T 문법이 있음
  - 윈도우에서는 Intel 문법 사용
  - 리눅스에서는 AT&T 문법 사용
  - Intel 문법에서는 목적지(destination)가 먼저 오고 원본(source)이 뒤에 위치
  - AT&T에서는 반대
  - Intel 문법에서 어셈블리어 명령 형식은 다음과 같음

Label :	MOV	AX,	BX ;	comment
라벨	작동 코드	제1피연산자	제2피연산자	설명

어셈블리어의 데이터 타입과 리틀 엔디안 방식

- 데이터 타입
  - 바이트(Byte)
    - 1바이트(8비트) 데이터 항목
  - 워드(Word)
    - 2바이트(16비트) 데이터 항목
  - 더블워드(Double word)
    - 4바이트(32비트) 데이터 항목

그림 2-11 어셈블리어 데이터 타입

어셈블리어의 데이터 타입과 리틀 엔디안 방식

- 리틀 엔디안 방식
  - 2개의 번지로 나누어 저장해야 하는 16비트 데이터(워드)의 경우 하위 바이트는 하위 번지에 상위 바이트는 상위 번지에 저장
  - hex 값 0x34F3을 1500번지에 저장하려면 하위 값 0xF3은 1500번지에,상위 값 0x34는 1501번지에 저장

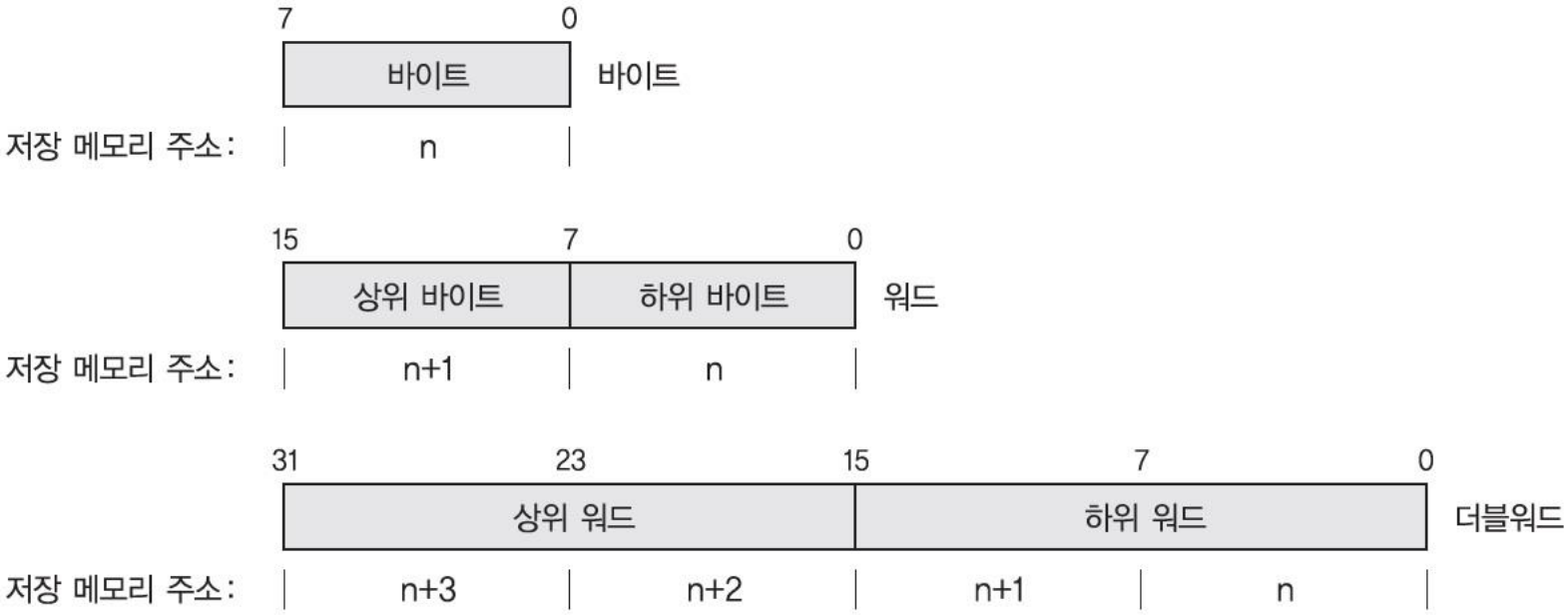


그림 2-12 어셈블리어의 데이터 타입별 메모리 저장 위치

어셈블리어의 주소 지정 방식

- 80x86 시스템은 메모리에 접근할 수 있도록 다양한 주소 지정 방식을 제공
  - 레지스터 주소 지정
  - 직접 메모리 주소 지정
  - 레지스터 간접 주소 지정
  - 인덱스 주소 지정
  - 베이스 인덱스 주소 지정
  - 변위를 갖는 베이스 인덱스 주소 지정

어셈블리어의 주소 지정 방식

- 레지스터 주소 지정
  - 레지스터의 주소 값을 직접 지정
  - 복사, 처리 속도 가장 빠름

```
MOV     DX,     BX
```

어셈블리어의 주소 지정 방식

- 직접 메모리 주소 지정
  - 가장 일반적인 주소 지정 방식
  - 보통 피연산자 하나가 메모리 위치를 참조하고 다른 하나는 레지스터를 참조
  - 예) DS:[8088h]와 DS:[1234h]는 각각 '세그멘트:오프셋' 형식의 메모리에 직접 접근하는 방식

```
MOV AL, DS:[8088h]
MOV DS:[1234h], DL
```

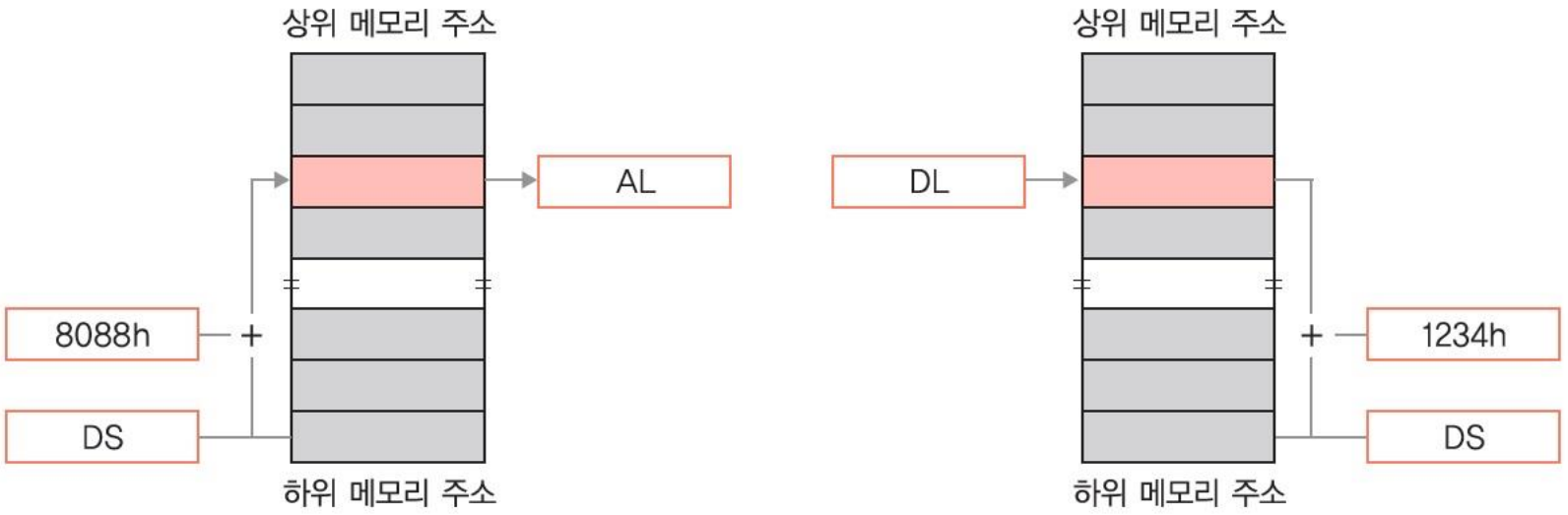


그림 2-13 직접 메모리 주소 지정 예

어셈블리어의 주소 지정 방식

- 레지스터 간접 주소 지정
  - '세그멘트:오프셋' 형식을 사용

```
MOV     AL, [BX]
MOV     AL, [BP]
```

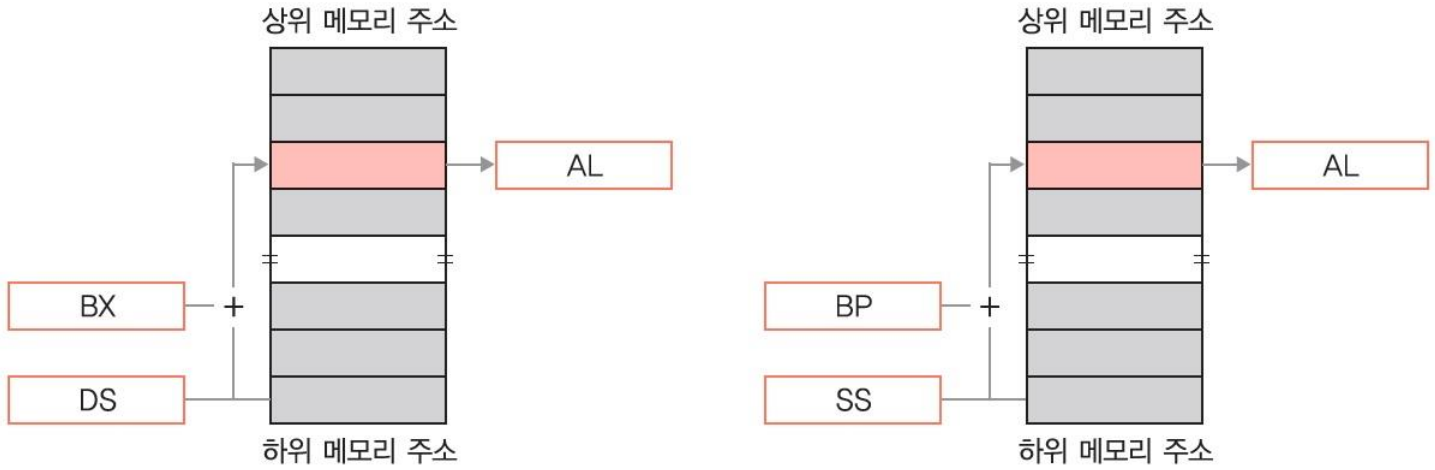


그림 2-14 레지스터 간접 주소 지정 예

  - 다음과 같이 기본이 아닌 세그먼트를 강제로 지정 할 수도 있음

```
MOV     AL, CS:[BX]
MOV     AL, DS:[BP]
```

어셈블리어의 주소 지정 방식

- 인덱스 주소 지정
  - 레지스터 간접 지정 방식에 변위가 더해진 메모리 주소 지정 방식
  - (예) 20h만큼 더해 메모리를 참조한 명령

```
MOV     AL, [BX+20h]
MOV     AL, [BP+20h]
```

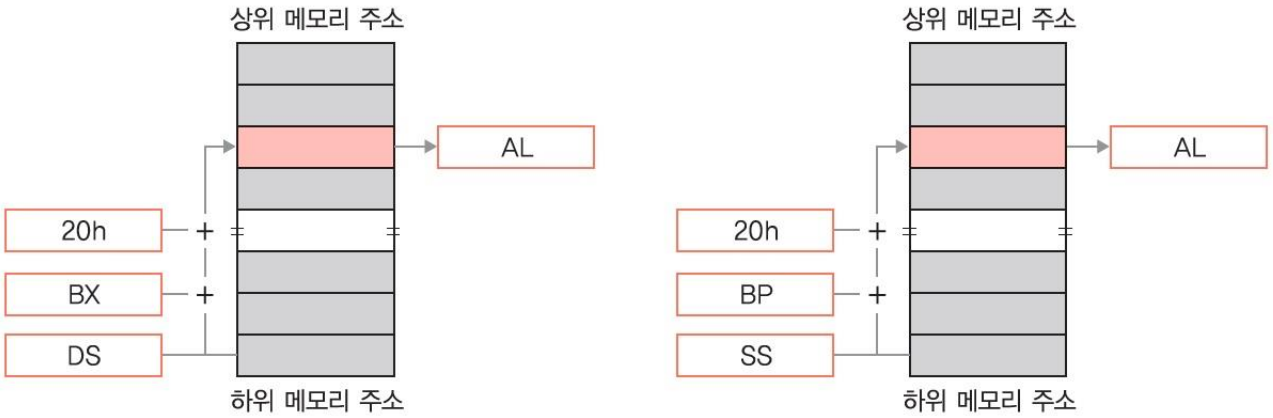


그림 2-15 인덱스 주소 지정 예

  - 다음과 같이 바꿔서 표현 할 수 있음

```
MOV     AL, 20h[BX]
MOV     AL, 20h[BP]
```

어셈블리어의 주소 지정 방식

- 베이스 인덱스 주소 지정
  - 실제 주소 생성 위해 베이스 레지스터(BX or BP)와 인덱스 레지스터(DI or SI)를 결합
  - 2차원 배열의 주소 지정에 사용

```
MOV     AL, [BX+SI]
MOV     AL, [BP+SI]
```

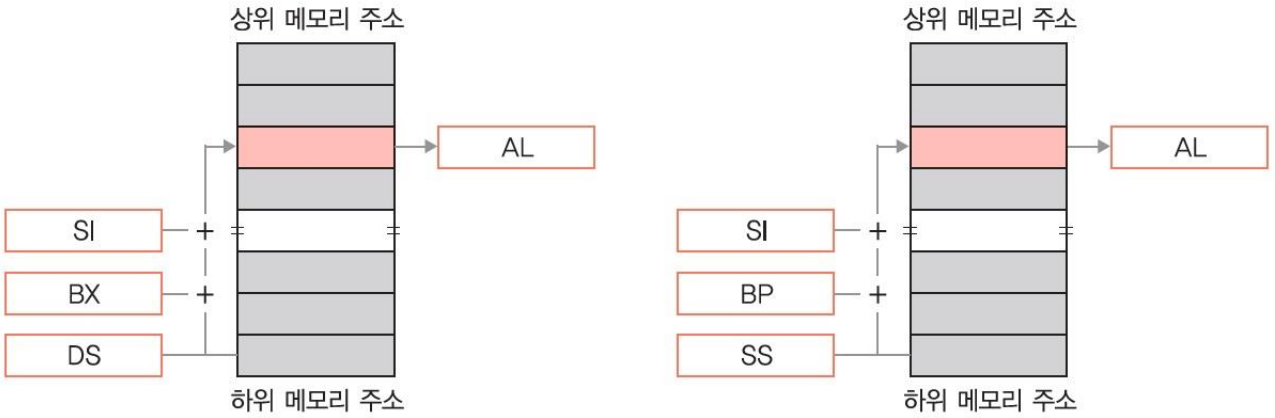


그림 2-16 베이스 인덱스 주소 지정 예

  - 다음과 같이 바꿔서 표현 할 수 있음

```
MOV     AL, [BX][SI]
MOV     AL, [BP][SI]
```

어셈블리어의 주소 지정 방식

- 변위를 갖는 베이스 인덱스 주소 지정
  - 베이스-인덱스의 변형으로 실제 주소 생성 위해 베이스 레지스터, 인덱스 레지스터, 변위 결합

```
MOV     AL, [BX+SI+20h]
MOV     AL, [BP+SI+20h]
```

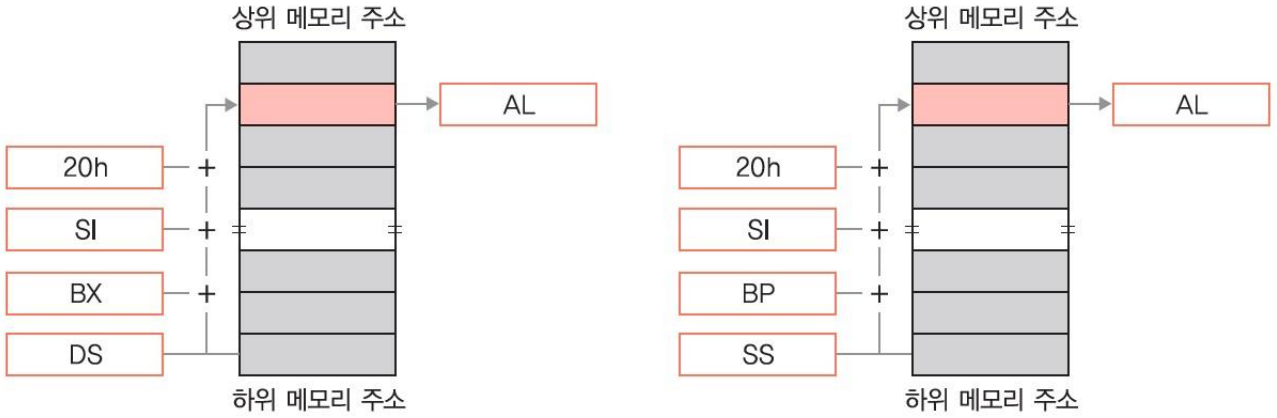


그림 2-17 변위를 갖는 베이스 인덱스 주소 지정 예

  - 다음과 같이 바꿔서 표현 할 수 있음

```
MOV     AL, [BX][SI][20h]
MOV     AL, [BP][SI][20h]
```

어셈블리어 기본 명령

- 산술 연산 명령 : 기본적인 정수 계산
  - ADD(Add) : 제1피연산자와 제2피연산자 값을 더해 제1피연산자에 저장

형식	ADD	[제1피연산자]	[제2피연산자]
사용 예	ADD	AL	4
	AL이 원래 3이었다면, 명령 실행 후 AL은 7이 된다.		

  - SUB(Subtract) : 제1피연산자에서 제2피연산자 값을 빼 제1피연산자에 저장

형식	SUB	[제1피연산자]	[제2피연산자]
사용 예	SUB	AL	4
	AL이 원래 7이었다면, 명령 실행 후 AL은 3이 된다.		

  - CMP(Compare) : 두 값을 비교하여 플래그 설정

어셈블리어 기본 명령

- 산술 연산 명령
  - 기타 산술 연산 명령

명령		설명
ADC	Add with Carry	캐리를 포함한 덧셈을 수행한다.
SBB	Subtraction with borrow	캐리를 포함한 뺄셈을 수행한다.
DEC	Decrement	피연산자 내용을 하나 감소시킨다.
NEG	Change Sign	피연산자의 2의 보수, 즉 부호를 반전한다.
INC	Increment	피연산자 내용을 하나 증가시킨다.
AAA	ASCII adjust for add	덧셈 결과의 AL 값을 UNPACK 10진수로 보정한다.
DAA	Decimal adjust for add	덧셈 결과의 AL 값을 PACK 10진수로 보정한다.
AAS	ASCII adjust for subtract	뺄셈 결과의 AL값을 UNPACK 10진수로 보정한다.

어셈블리어 기본 명령

- 산술 연산 명령
  - 기타 산술 연산 명령

명령		설명
DAS	Decimal adjust for subtract	뺄셈 결과의 AL값을 PACK 10진수로 보정한다.
MUL	Multiply(Unsigned)	AX와 피연산자의 곱셈 결과를 AX 또는 DX:AX에 저장한다.
IMUL	Integer Multiply(Signed)	부호화된 곱셈을 수행한다.
AAM	ASCII adjust for Multiply	곱셈 결과의 AX 값을 UNPACK 10진수로 보정한다.
DIV	Divide(Unsigned)	AX 또는 DX:AX 내용을 피연산자로 나눈다. 몫은 AL 또는 AX에 저장하고, 나머지는 AH 또는 DX에 저장한다.
IDIV	Integer Divide(Signed)	부호화된 나눗셈
AAD	ASCII adjust for Divide	나눗셈 결과 AX 값을 UNPACK 10진수로 보정한다.
CBW	Convert byte to word	AL의 바이트 데이터를 부호 비트를 포함하여 AX 워드로 확장한다.
CWD	Convert word to double word	AX의 워드 데이터를 부호를 포함하여 DX:AX의 더블 워드로 변환한다.

어셈블리어 기본 명령

- 데이터 전송 명령
  - 메모리, 범용 레지스터, 세그먼트 레지스터로 참조되는 주소에 들어 있는 데이터 전송
  - MOV(Move)
    - 데이터를 이동할 때 사용
    - BP의 현재 값이 0x10000004라면, BP+8은 0x1000000C
    - 0x1000000C에 있는 값이 1024므로 AX에는 1024가 입력

형식	MOV	[제1피연산자]	[제2피연산자]
사용 예	MOV	AX	[BP+8]
	BP의 주소에 8 더해진 주소에 있는 데이터 값을 AX에 대입한다.		

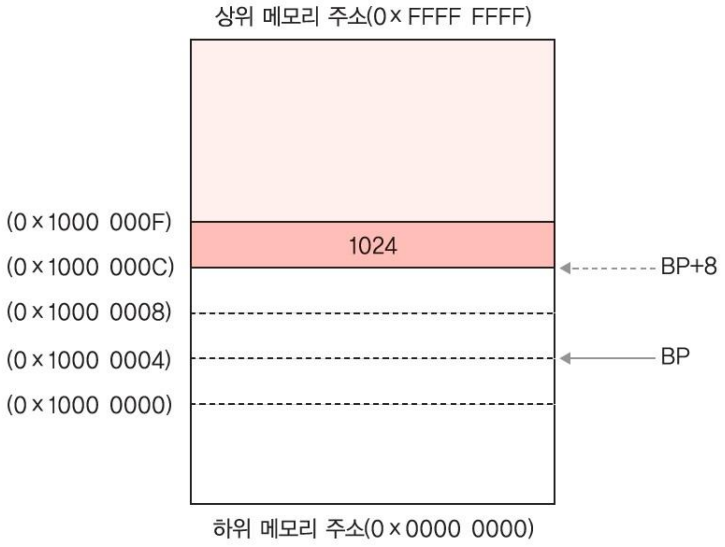


그림 2-18 MOV 명령을 실행할 때 스택 동작

어셈블리어 기본 명령

- 데이터 전송 명령
  - PUSH(Push)
    - 스택에 데이터를 삽입할 때 사용
    - 아래 그림과 같이 스택은 커지고, 스택 포인터(Stack Pointer)는 데이터 크기 만큼 감소

형식	PUSH	[제1피연산자]

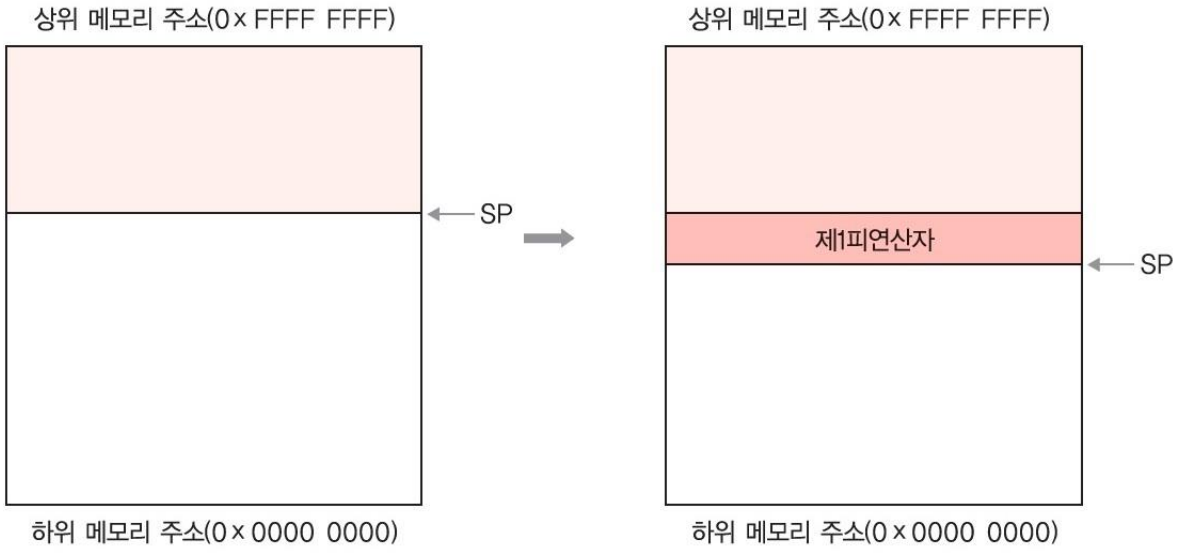


그림 2-19 PUSH 명령을 실행할 때 스택 동작

어셈블리어 기본 명령

- 데이터 전송 명령
  - POP(Pop)
    - 스택에서 데이터를 삭제할 때 사용
    - 스택에서 삭제된 명령은 반환 값으로 받아 사용할 수 있음
    - 아래 그림과 같이 스택 포인터는 삭제하는 데이터 크기만큼 증가

형식	POP	[제1피연산자]

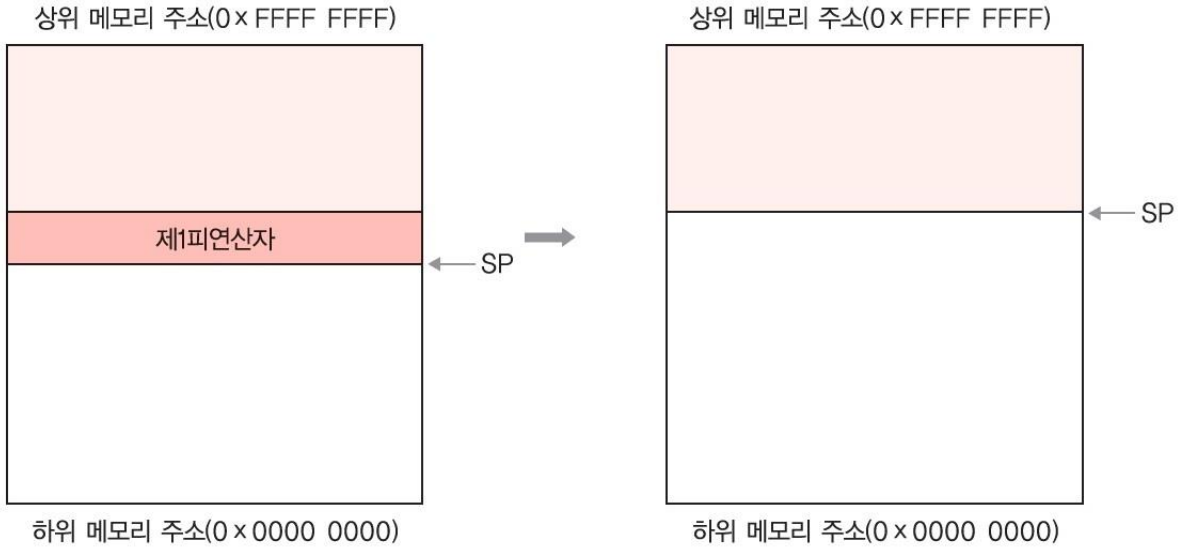


그림 2-20 POP 명령을 실행할 때 스택 동작

어셈블리어 기본 명령

- 데이터 전송 명령
  - LEA(Load effective address to register)
    - 데이터 값을 이동할 때 사용
    - MOV 명령과 LEA 명령에서 제1피연산자는 공통적으로 데이터 이동의 목적지
    - 제2피연산자에 'BP+4'가 있을 경우 MOV 명령은 'BP+4'를 하나의 주소 값으로 처리하지만 LEA 명령은 'BP'만 주소 값으로 인식 '+4'는 BP 주소 값에 대한 추가 연산으로 처리

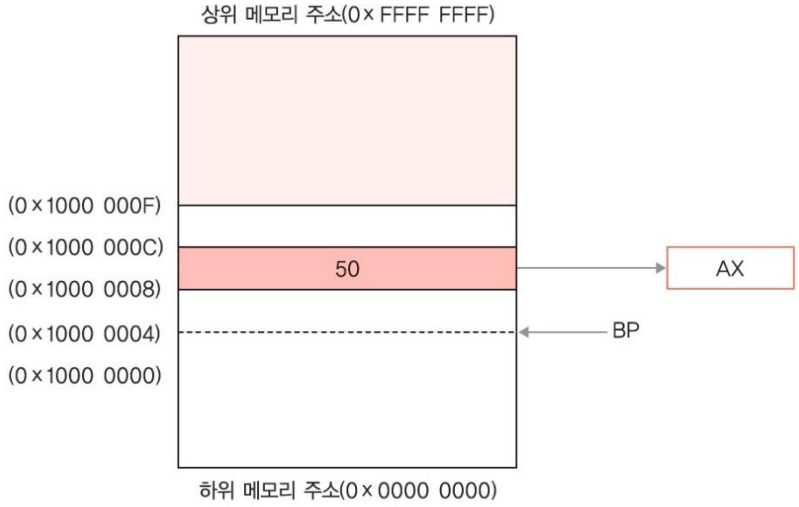


그림 2-21 LEA 명령을 실행할 때 스택 동작

형식	MOV [제1피연산자] [제2피연산자]
사용 예	LEA AX [BP+4]
	BP의 현재 값이 0×10000004라면, MOV 명령처럼 BP+4인 0×10000008의 주소 값을 가져오는 것이 아니라 0×10000004의 주소에 있는 값에 4를 더해 AX에 대입한다.

어셈블리어 기본 명령

- 데이터 전송 명령
  - 기타 데이터 전송 명령

명령		설명
XCHG	Exchange Register/ Memory with Register	첫 번째 피연산자와 두 번째 피연산자를 바꾼다.
IN	Input from AL/ AX to Fixed port	피연산자로 지시된 포트로 AX에 데이터를 입력한다.
OUT	Output from AL/AX to Fixed port	피연산자가 지시한 포트로 AX의 데이터를 출력한다.
XLAT	Translate byte to AL	BX:AL이 지시한 테이블의 내용을 AL로 로드한다.
LDS	Load Pointer to DS	LEA 명령과 유사한 방식으로 다른 DS 데이터의 주소의 내용을 참조 시 사용한다.
LES	Load Pointer to ES	LEA 명령과 유사한 방식으로 다른 ES 데이터의 주소의 내용을 참조 시 사용한다.
LAHF	Load AH with Flags	플래그의 내용을 AH의 특정 비트로 로드한다.
SAHF	Store AH into Flags	AH의 특정 비트를 플래그 레지스터로 전송한다.
PUSHF	Push Flags	플래그 레지스터의 내용을 스택에 삽입한다.
POPF	Pop Flags	스택에 저장되어 있던 플래그 레지스터 값을 삭제한다.

어셈블리어 기본 명령

- 논리 명령
  - 연산 부호가 논리 연산을 지정하는 명령으로 자리옮김, 논리합(OR), 논리곱(AND), 기호 변환 등이 있음
  - AND(And)
    - 대응되는 비트가 둘 다 1일 때만 결과가 1이고, 그 이외는 모두 0

<table>
<tr><th>형식</th><th>AND [제1피연산자] [제2피연산자]</th></tr>
<tr><td rowspan="2">사용 예</td><td>AND AX 10h</td></tr>
<tr><td>AX 값이 0x08h라면 이를 이진수로 표현하면 1000이 된다.
0x10h는 1010이므로, 명령을 수행한 뒤에 AX 값은 1000이 된다.
<table>
<tr><th>AX</th><th>1</th><th>0</th><th>0</th><th>0</th></tr>
<tr><td>0×10h</td><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>AND 연산 결과</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
</table>
</td></tr>
</table>

어셈블리어 기본 명령

- 논리 명령
  - OR(Or)
    - 대응되는 비트 중 하나만 1이어도 결과가 1이고, 둘 다 0인 경우에만 0

<table>
<tr><th>형식</th><th>OR [제1피연산자] [제2피연산자]</th></tr>
<tr><td rowspan="2">사용 예</td><td>OR AX 10h</td></tr>
<tr><td>AX 값이 0x08h라면 이를 이진수로 표현하면 1000이 된다.
0x10h는 1010이므로, 명령을 수행한 뒤에 AX 값은 1010이 된다.
<table>
<tr><th>AX</th><th>1</th><th>0</th><th>0</th><th>0</th></tr>
<tr><td>0×10h</td><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>OR 연산 결과</td><td>1</td><td>0</td><td>1</td><td>0</td></tr>
</table>
</td></tr>
</table>

어셈블리어 기본 명령

- 논리 명령
  - XOR(Exclusive Or)
    - 대응되는 비트 중에서 한 비트가 1이고 다른 비트가 0이면 1 두 개의 비트가 모두 0 또는 1일 때 0

<table>
<tr><th>형식</th><th>XOR [제1피연산자] [제2피연산자]</th></tr>
<tr><td rowspan="2">사용 예</td><td>XOR AX 10h</td></tr>
<tr><td>AX 값이 0x08h라면 이를 이진수로 표현하면 1000이 된다.
0x10h는 1010이므로, 명령을 수행한 뒤에 AX 값은 0010이 된다.
<table>
<tr><td>AX</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0×10h</td><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>OR 연산 결과</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
</table>
</td></tr>
</table>

어셈블리어 기본 명령

- 논리 명령
  - NOT(Invert)
    - 피연산자의 1의 보수를 구하는 작동 코드로, 각 비트를 반전

<table>
<tr><th>형식</th><th>NOT [제1피연산자]</th></tr>
<tr><td rowspan="2">사용 예</td><td>NOT AX</td></tr>
<tr><td>AX 값이 0x08h라면 이를 이진수로 표현하면 1000이 된다.
명령을 수행한 뒤에 AX 값은 0111이 된다.
<table>
<tr><td>AX</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>NOT 연산 결과</td><td>0</td><td>1</td><td>1</td><td>1</td></tr>
</table>
</td></tr>
</table>

어셈블리어 기본 명령

- 논리 명령
  - Test(And function to flags, no result)
    - 데이터의 두 값 비교할 때 사용, 데이터의 변경 없이 단순 비교

형식	TEST [제1피연산자] [제2피연산자]
사용 예	TEST AL 00001001b
	AL 레지스터에 비트 0과 비트 3이 둘 다 0인지 확인하고 싶으면 제2피연산자에 비트 0(오른쪽의 첫 번째 비트)과 비트 3(오른쪽부터 네 번째 비트)에 1을 적어주고 나머지에는 0을 적어준 후 TEST 명령을 실행한다. AL의 비트 0과 비트 3이 모두 0이라면 ZF(Zero Flag)가 세트된다. 비트 0와 비트 3 중 하나라도 1이면 ZF는 클리어된다. AX가 0x08h라면 비트 3이 1이므로 ZF는 클리어된다.

AL	1	0	0	0
00001001b	1	0	0	1

어셈블리어 기본 명령

- 논리 명령
  - 기타 논리 명령

명령		설명
SHL / SAL	Shift Left/arithmetic Left	왼쪽으로 피연산자만큼 자리 이동
SHR /SAR	Shift Right / Shift arithmetic Right	오른쪽으로 피연산자만큼 자리 이동
ROL	Rotate Left	왼쪽으로 피연산자만큼 회전 이동
ROR	Rotate Right	오른쪽으로 피연산자만큼 회전 이동
RCL	Rotate through carry left	자리올림(Carry)을 포함하여 왼쪽으로 피연산자만큼 회전 이동
RCR	Rotate through carry Right	자리올림(Carry)을 포함하여 오른쪽으로 연산자만큼 회전이동

어셈블리어 기본 명령

- 스트링 명령
  - 바이트로 구성된 데이터(Strings of Bytes)를 메모리에서 가져오거나 메모리로 전송
  - REP(Repeat)
    - ADD나 MOVS 같은 작동 코드 앞에 위치, CX가 0이 될 때까지 뒤에 오는 스트링 명령을 반복

형식	REP	작동코드	-

어셈블리어 기본 명령

- 스트링 명령
  - MOVS(Move String)
    - 바이트나 워드, 더블워드를 옮기는 명령
    - MOVSB, MOVSW, MOVSD 가 있음
    - DS:SI가 지시한 메모리 데이터를 ES :DI가 지시한 메모리로 전송

형식	MOVSB
사용 예	CLD LEA SI, String_1 LEA DI, String_2 MOV CX, 384 REP MOVSB
	- CLD (Clear Direction) : 디렉션 플래그를 지움 - LEA SI, String_1 : String 1의 주소 값을 SI(Source Index)에 저장 - LEA DI, String_2 : String 2의 주소 값을 DI(Destination Index)에 저장 - MOV CX, 384 : CX에 384 저장 - REP MOVSB : SI로부터 DI까지 CX가 0이 될 때까지 1바이트씩 복사

어셈블리어 기본 명령

- 스트링 명령
  - 기타 스트링 명령

명령문		설명
CMPS	Compare String	DS: SI와 ES: DI의 내용을 비교한 결과에 따라 플래그를 설정한다.
SCAS	Scan String	AL 또는 AX와 ES:DI가 지시한 메모리 내용을 비교한 결과에 따라 플래그를 설정한다.
LODS	Load String	SI 내용을 AL 또는 AX로 로드한다.
STOS	Store String	AL 또는 AX를 ES: DI가 지시하는 메모리에 저장한다.

어셈블리어 기본 명령

- 제어 전송 명령
  - 점프(분기, Jump), 조건 점프(조건 분기, Conditional Jump), 루프(Loop), 호출(Call)과 리턴(Return) 연산 등으로 프로그램의 흐름 제어
  - JMP(Unconditional Jump)
    - 대표적인 점프 명령 프로그램을 실행할 주소 또는 라벨로 이동

형식	JMP [제1피연산자]
사용 예	JMP 100h
	주소로 직접 지정한 100h번지로 점프한다.
	string: MOV CX, 384 ... JMP string
	라벨로 JMP를 지정하는 경우다.

어셈블리어 기본 명령

- 제어 전송 명령
  - 조건부 점프 명령

명령		점프 조건	설명
JC	carry flag set	CF = 1	CF 값이 1이면 점프
JNC	not carry flag set	CF = 0	CF 값이 0 이면 점프
JE/JZ	Equal / Zero	ZF = 1	결과가 0이면 점프
JA/JNBE	above/not below nor equal	CF = 0 and ZF = 0	결과가 크면 점프(부호화 안된 수)
JAE/JNB	above or equal/not below	CF = 0	결과가 크거나 같으면 점프(부호화 안된 수)
JB/JNAE	below/not above nor equal	CF = 1	결과가 작으면 점프(부호화 안된 수)
JL/JNGE	less/not greater nor equal	SF ! = OF	결과가 작으면 점프(부호화된 수)
JBE/JNA	below or equal/not above	(CF or ZF) = 1	결과가 작거나 같으면 점프(부호화 안된 수)
JG/JNLE	greater/not less nor equal	ZF = 0 and SF = OF	결과가 크면 점프(부호화 안된 수)
JGE/JNL	greater or equal/not less	SF = OF	결과가 크거나 같으면 점프(부호화된 수)

어셈블리어 기본 명령

- 제어 전송 명령
  - 조건부 점프 명령

명령		점프 조건	설명
JLE/JNG	less or equal/not greater	ZF = 1 or SF ! = OF	결과가 작거나 같으면 점프(부호화 안된 수) 결과가 0이 아니면 점프
JNE/JNZ	not equal/not zero	ZF = 0	JNOnot overflowOF=0오버플로우가 아닌 경우 점프
JNP/JPO	not parity/parity odd	PF = 0	PF가 1이면 점프
JNS	not sign	SF = 0	SF가 1이면 점프
JO	overflow	OF = 1	오버플로우 발생 시 점프
JP/JPE	parity/parity even	PF = 1	PF가 1이면 점프
JS	Sign	SF = 1	SF가 1이면 점프
JCXZ	CX Zero	CX = 0	CX가 0이면 점프

* above, below : 부호 없는 두 수의 크기 관계

* greater, less : 부호 있는 두 수의 크기 관계

어셈블리어 기본 명령

- 제어 전송 명령
  - CALL(Call)
    - JMP처럼 함수 호출할 때 사용, 제1피연산자에 라벨을 지정
    - 리턴 주소로 IP(Instruction Pointer) 주소 백업
    - ‘PUSH EIP + JMP’와 같은 의미

형식	CALL	[제1피연산자]

  - RET(Return from CALL)
    - 함수에서 호출한 곳으로 돌아갈 때 사용하는 명령
    - ‘POP EIP’와 같은 의미

```
RET
```

어셈블리어 기본 명령

- 제어 전송 명령
  - CALL과 RET의 예 (AX에는 0x08h 값이 저장되어 있다고 가정)

```
        CALL    SUBR
        ADD     AX,     10h
        ...
SUBR :  INC     AX
        ....
        RET
```

  - SUBR 함수 호출하면, SUBR 라벨이 있는 곳에서 RET까지 실행. INC AX가 있으므로 AX는 0x09h
  - RET 명령이 실행되면 CALL 다음 라인인 'ADD AX, 10h' 실행
  - AX는 0x19h

어셈블리어 기본 명령

- 제어 전송 명령
  - LOOP(Loop CX times)
    - 문장들의 블록을 지정된 횟수만큼 반복
    - CX는 자동적으로 카운터로 사용되며 루프 반복할 때마다 감소

형식	LOOP [제1피연산자]

사용 예	MOV AX, 0
	MOV CX, 5
	
	L1 : INC AX
	LOOP L1

	제1피연산자는 라벨이 된다. 예에서 L1이 CX의 숫자만큼 5번 회전하므로, 결과적으로 AX는 5가 된다.

어셈블리어 기본 명령

- 제어 전송 명령
  - INT(Interrupt)
    - 인터럽트가 호출되면 CS:IP(Code Segment : Instruction Pointer)와 플래그를 스택에 저장
    - 그 인터럽트에 관련된 서브 루틴이 실행

형식	INT	[제1피연산자]

어셈블리어 기본 명령

- 프로세스 제어 명령
  - STC(Set Carry)
    - 피연산자 없이 사용, EFLAGS 레지스터의 CF 값을 세팅
  - NOP(No Operation)
    - 아무 의미 없는 명령, 일종의 빈 칸을 채우려고 사용
  - 기타 프로세스 제어 명령

명령		설명
CLC	Clear Carry	캐리 플래그를 클리어한다.
CMC	Complement Carry	캐리 플래그를 반전한다.
HLT	Halt	정지한다.
CLD	Clear Direction	디렉션 플래그를 클리어한다.
CLI	Clear Interrupt	인터럽트 플래그를 클리어한다.
STD	Set Direction	디렉션 플래그를 세트한다.
STI	Set Interrupt	인터럽트 인에이블 플래그를 세트한다.
WAIT	Wait	프로세스를 일시 정지 상태로 한다.
ESC	Escape to External device	종료 명령이다.

스택을 이용한 명령 처리 과정

스택이란?

- 스택(stack)
  - 접시를 쌓듯이 자료를 차곡차곡 쌓아 올린 형태의 자료구조

그림 5-1 스택의 개념 예

스택이란?

- 스택(stack)
  - 스택에 저장된 원소는 top으로 정한 곳에서만 접근 가능
    - top의 위치에서만 원소를 삽입하므로, 먼저 삽입한 원소는 밑에 쌓이고, 나중에 삽입한 원소는 위에 쌓이는 구조
    - 마지막에 삽입한 원소는 맨 위에 쌓여 있다가 가장 먼저 삭제됨
      - 후입선출 구조 (LIFO, Last-In-First-Out)

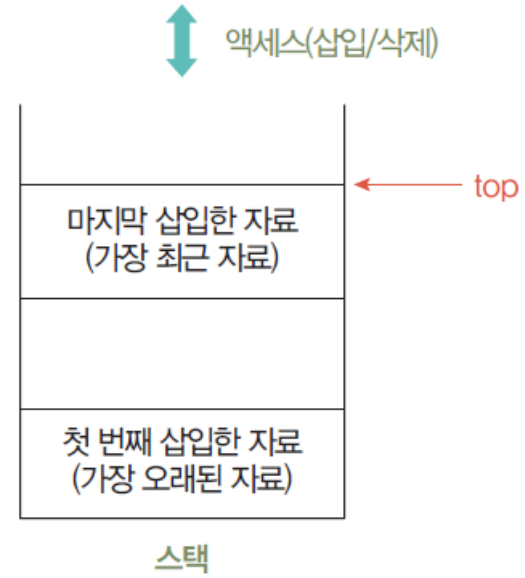


그림 5-2 스택의 구조

후입선출 구조의 예

- 연탄 아궁이
  - 연탄을 하나씩 쌓으면서 아궁이에 넣으므로 마지막에 넣은 3번 연탄이 가장 위에 쌓여 있음
  - 연탄을 아궁이에서 꺼낼 때에는 위에서부터 하나씩 꺼내야 하므로 마지막에 넣은 3번 연탄을 가장 먼저 꺼내게 됨

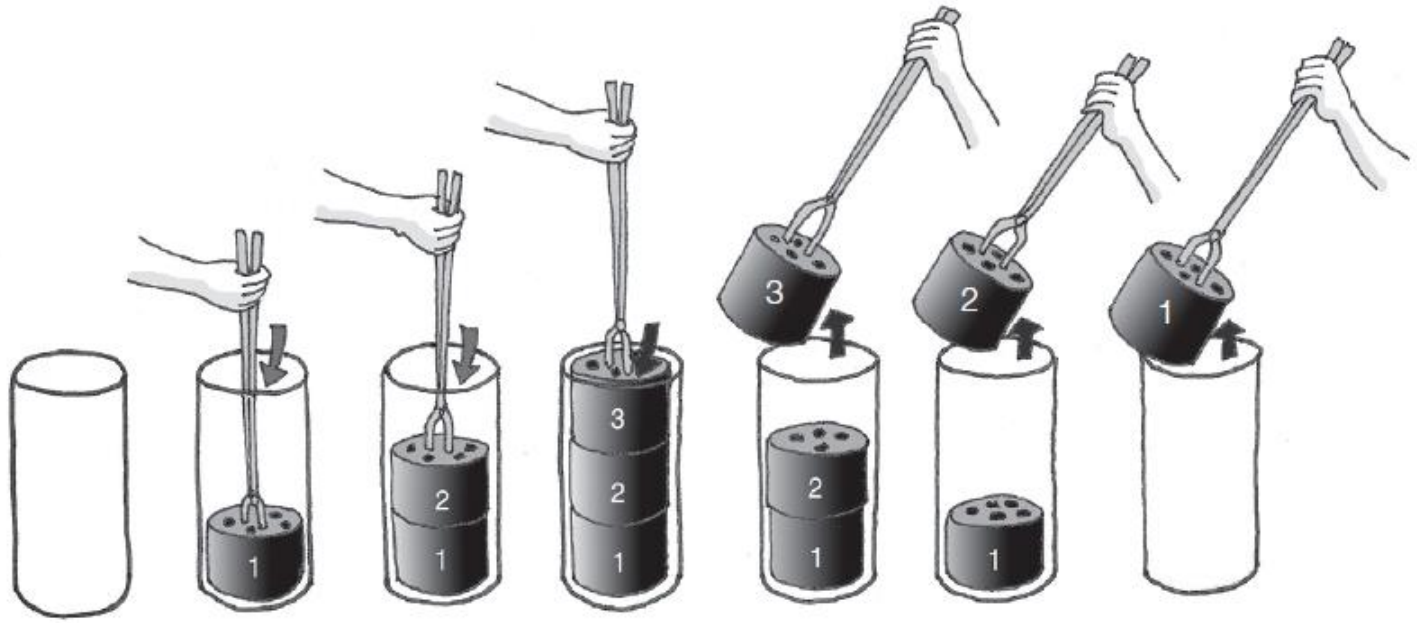


그림 5-3 스택의 LIFO 구조 예 : 연탄 아궁이

스택의 연산

- 스택에서의 삽입 연산
  - push
- 스택에서의 삭제 연산
  - pop

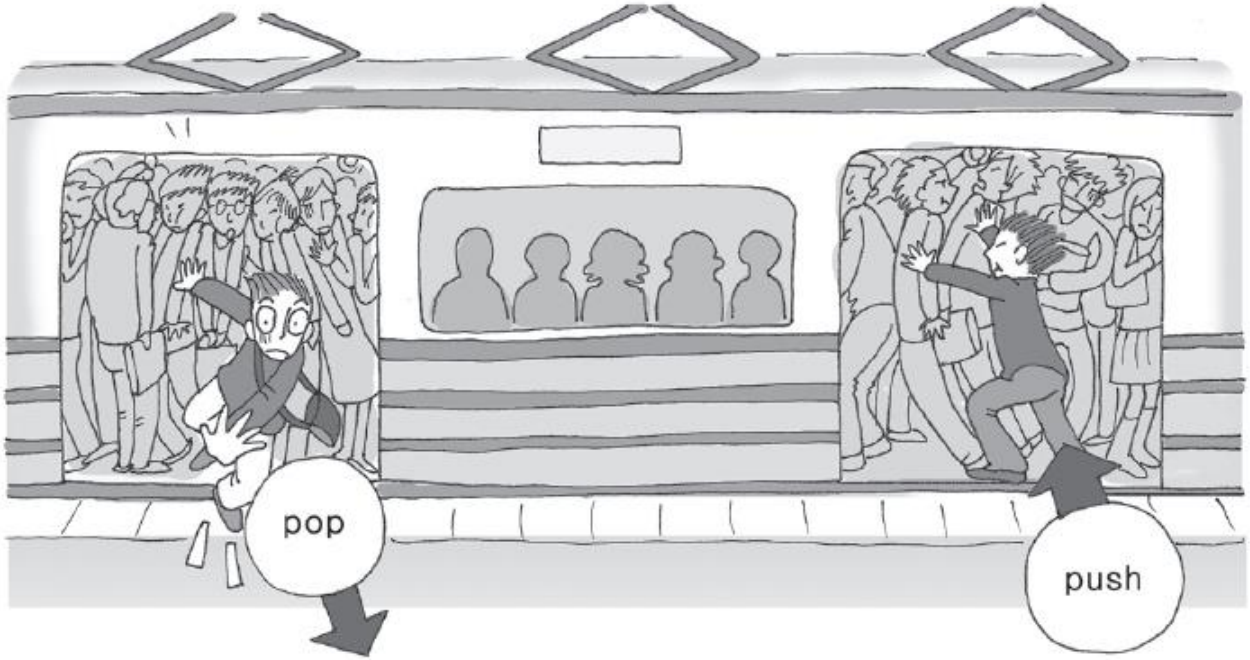


그림 5-5 만원 전철에서의 pop과 push

스택의 연산

- 스택에서의 원소 삽입/삭제 과정
  - 공백 스택에 원소 A, B, C를 순서대로 삽입하고 하나씩 삭제하는 연산 과정 동안의 스택 변화

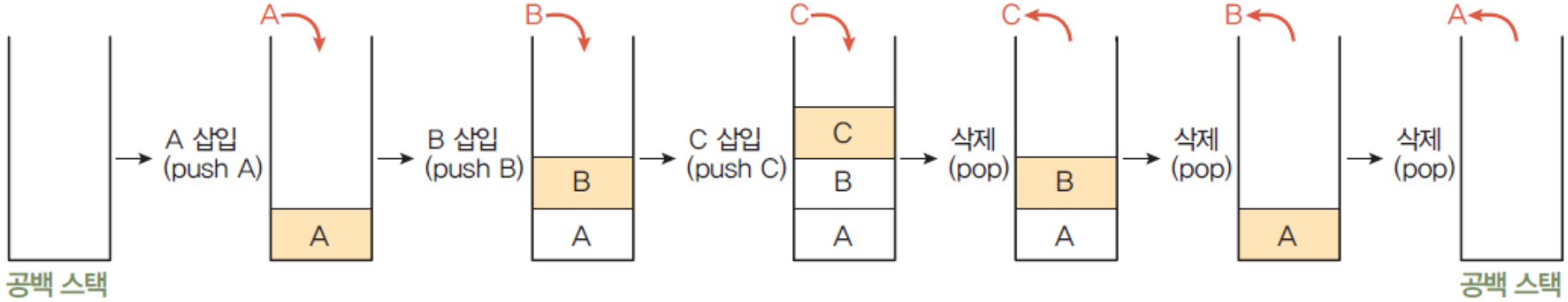


그림 5–6 스택의 데이터 삽입(push)과 삭제(pop) 과정

시스템 스택

- 프로그램에서의 호출과 복귀에 따른 수행 순서를 관리
  - 가장 마지막에 호출된 함수가 가장 먼저 실행을 완료하고 복귀하는 후입선출 구조이므로, 후입선출 구조의 스택을 이용하여 수행순서 관리
  - 함수 호출이 발생하면 호출한 함수 수행에 필요한 지역변수, 매개변수 및 수행 후 복귀할 주소 등의 정보를 스택 프레임(stack frame)에 저장하여 시스템 스택에 삽입
  - 함수의 실행이 끝나면 시스템 스택의 top 원소(스택 프레임)를 삭제(pop)하면서 프레임에 저장되어있던 복귀주소를 확인하고 복귀
  - 함수 호출과 복귀에 따라 이 과정을 반복하여 전체 프로그램 수행이 종료되면 시스템 스택은 공백 스택이 됨

시스템 스택

- 함수 호출과 복귀에 따른 전체 프로그램의 수행 순서

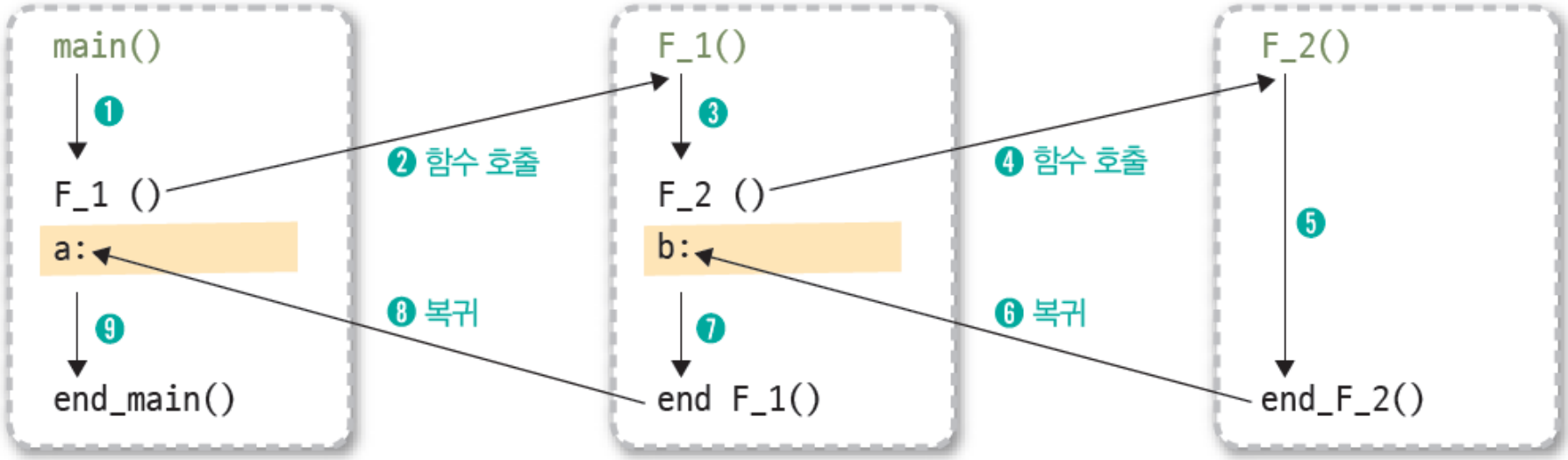


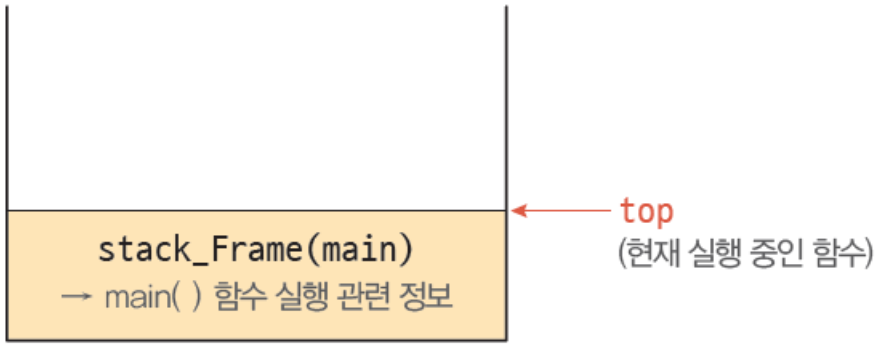
그림 5-9 함수 호출과 복귀에 따른 전체 프로그램 수행 순서

시스템 스택

① main( )함수 실행 시작

- 시작하면 main( )함수가 호출되어 실행
- main( ) 함수 시작과 관련된 정보를 스택 프레임에 저장, 시스템 스택에 삽입

```
push(System_stack, stack_Frame(main));
```

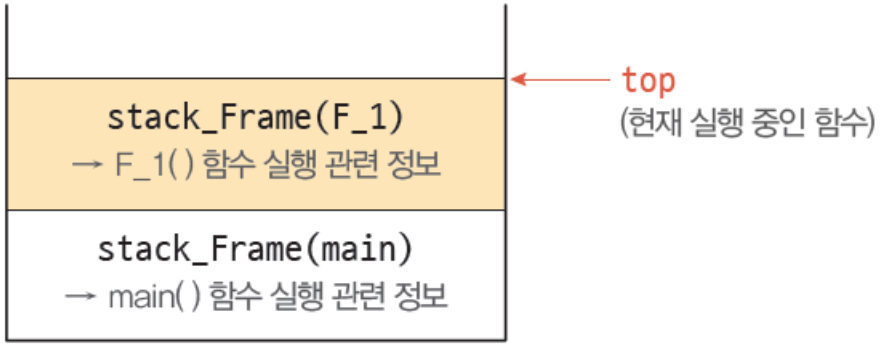


시스템 스택

② F_1( ) 함수 호출

- main() 함수 실행 중 F_1 () 함수 호출을 만나면 함수 호출과 복귀에 필요한 정보를 스택 프레임에 저장, 시스템 스택에 삽입, 호출된 함수인 F_1 () 함수로 이동
- 이 때 스택 프레임에는 호출된 함수의 수행이 끝나고 main() 함수로 복귀할 주소 a 를 저장

```
push(System_stack, stack_Frame(F_1));
```



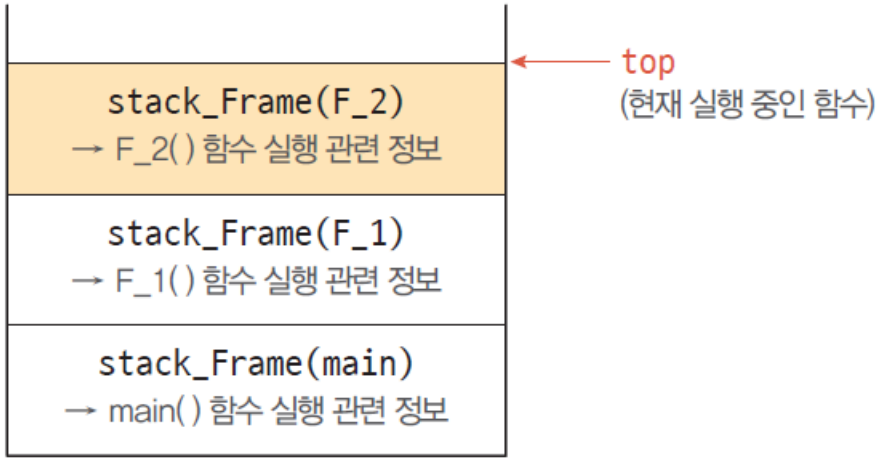
시스템 스택

③ 호출된 함수 F_1() 함수 실행

④ F_2( ) 함수 호출

- F_1() 함수 실행 중에 F_2() 함수 호출을 만나면 다시 함수 호출과 복귀에 필요한 정보를 스택 프레임에 저장하여 시스템 스택에 삽입하고, 호출된 함수인 F_2() 함수를 실행
- 스택 프레임에는 F_1() 함수로 복귀할 주소 b를 저장

```
push(System_stack, stack_Frame(F_2));
```



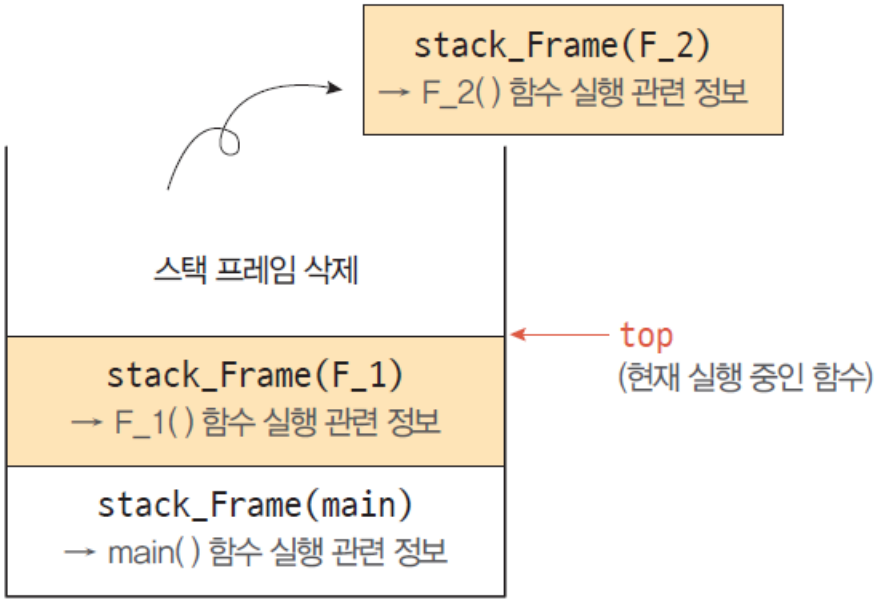
시스템 스택

⑤ 호출된 함수 F_2() 함수 실행

⑥ F_2( ) 함수 실행 종료, F_1( ) 함수로 복귀

- F_2( ) 함수 실행이 끝나면 F_2( ) 함수를 호출했던 이전 위치로 복귀하여 이전 함수 F_1( )의 작업을 계속해야 함
- 시스템 스택의 top에 있는 스택 프레임을 pop하여 정보를 확인하고 복귀 및 작업 전환 실행함

```
pop(System_stack);
```



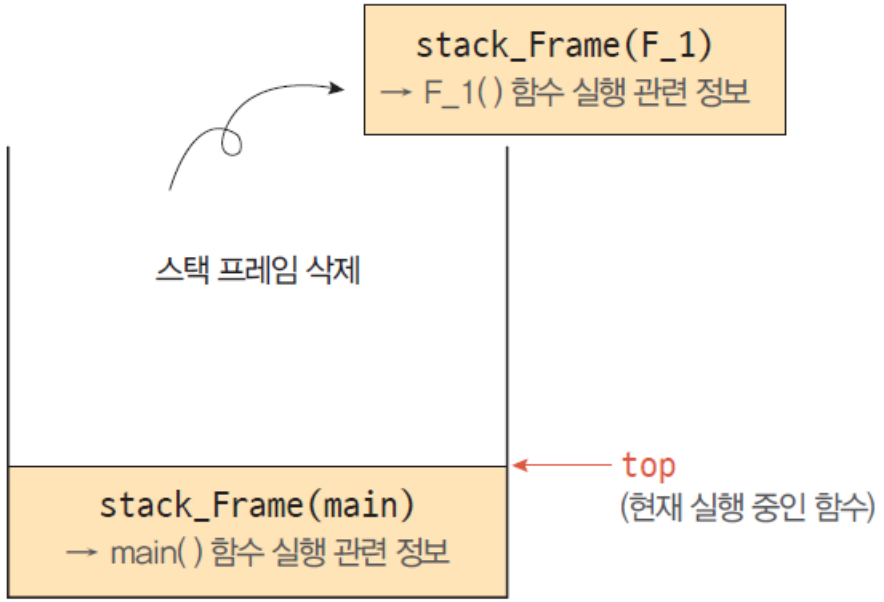
시스템 스택

⑦ F_1( ) 함수로 복귀하여 F_1( ) 함수의 나머지 부분 실행

⑧ F_1( ) 함수 실행 종료, main( ) 함수로 복귀

– 스택의 top에 있는 스택 프레임을 pop하여 정보를 확인하고 복귀 및 작업 전환을 실행

```
pop(System_stack);
```

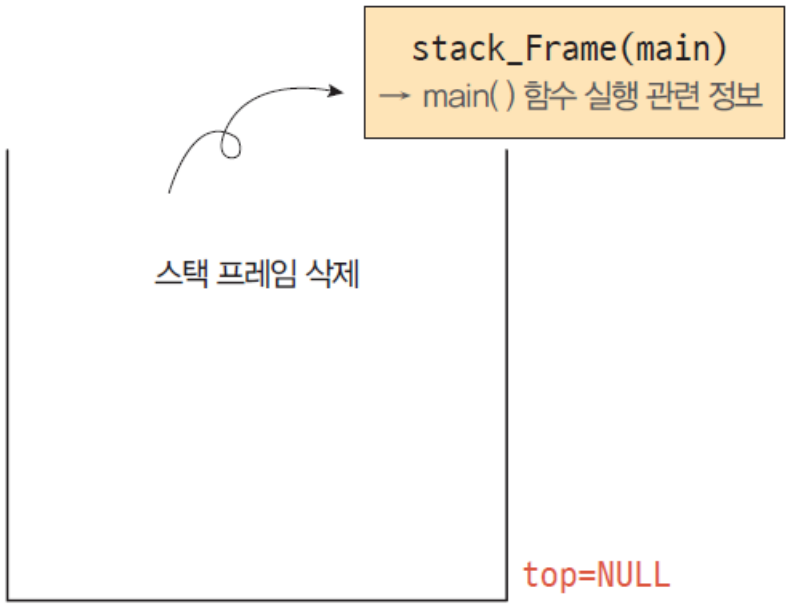


시스템 스택

⑨ main( ) 함수 실행 완료(전체 프로그램 실행 완료)

– 정상적인 함수 호출과 복귀가 모두 완료되었으므로 시스템 스택은 공백이 됨

```
pop(System_stack);
```



프로그램 실행 과정에 따른 스택의 동작 이해하기

- 어셈블리어의 각 실행 단계별로 스택의 동작을 살펴보는 과정을 통해 스택에서 어셈블리어 처리하는 과정을 이해
- 리눅스에서는 어셈블리어가 AT&T 문법으로 사용되어 제1피연산자와 제2피연산자의 위치가 반대
- 실습 환경
  - Redhat Linux 6.2
- 필요 프로그램
  - gcc, gdb

sample.c의 어셈블리 코드 획득 (1)

- sample.c

```
void main()
{
	int c;
	c=function(1, 2);
}

int function(int a, int b)
{
	char buffer[10];
	a=a+b;
	return a;
}
```

sample.c의 어셈블리 코드 획득 (2)

- 컴파일

```
gcc -S -o sample.a sample.c
vi sample.a
```

```
.file           "sample.c"
                .version        "01.01"
gcc2_compiled.:
.text
                .align 4
.globl main
                .type           main,@function
main:
                pushl           %ebp                    ❶
                movl            %esp,%ebp               ❷
                subl            $4,%esp                 ❸
                pushl           $2                      ❹
                pushl           $1                      ❺
                call            function                ❻
                addl            $8,%esp                 ⓱
                movl            %eax,%eax               ⓲
```

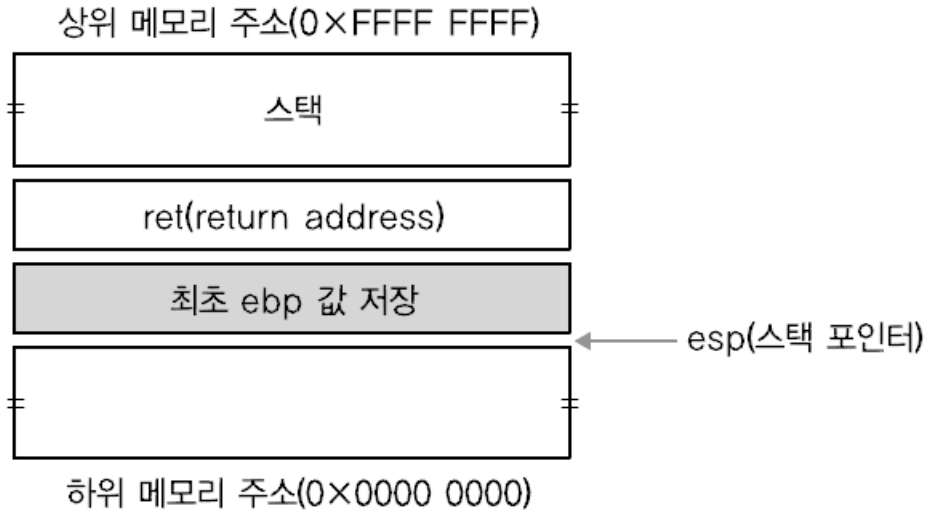
sample.c의 어셈블리 코드 획득 (3)

```
        movl %eax,-4(%ebp)                                  ⓳
.L1:
        leave                                               ⓴
        ret                                                 ■
.Lfe1:
        .size       main,.Lfe1-main
        .align 4
.globl function
        .type       function,@function
function:
        pushl       %ebp                                    ❼
        movl        %esp,%ebp                               ❽
        subl        $12,%esp                                ❾
        movl        12(%ebp),%eax                           ❿
        addl        %eax,8(%ebp)                            ⓫
        movl        8(%ebp),%edx                            ⓬
        movl        %edx,%eax                               ⓭
        jmp .L2                                             ⓮
        .p2align 4,,7
.L2:
        leave                                               ⓯
        ret                                                 ⓰
.Lfe2:
        .size       function,.Lfe2-function
        .ident      "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

기본 코드 분석

❶ pushl %ebp

- 최초의 프레임 포인터(ebp) 값 스택에 저장
- ebp 바로 전에 ret가 저장
- ebp는 함수 시작 전의 기준점이 됨
- 스택에 저장된 ebp를 SFP(Saved Frame Pointer)라고 부름
- ret(return address)에는 함수 종료 시 점프할 주소 값 저장

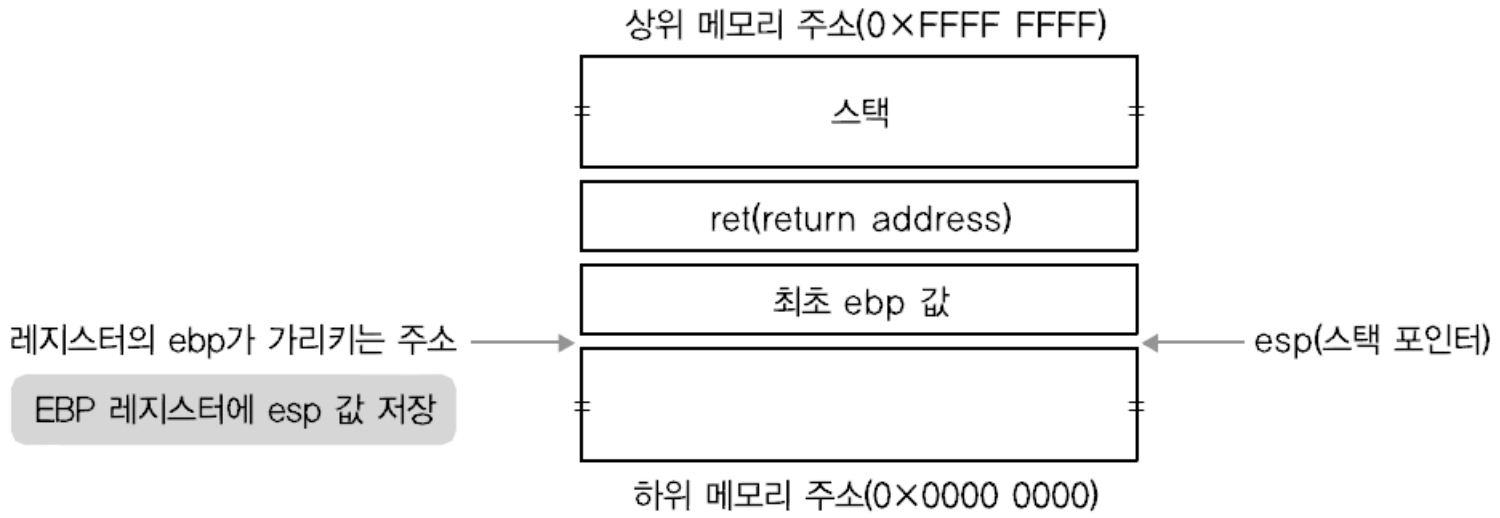


[그림 2-27] pushl %ebp 실행 시 스택의 구조

기본 코드 분석

❷ movl %esp, %ebp

- 현재의 esp 값을 EBP 레지스터에 저장
- push %ebp와 mov %esp, %ebp는 새로운 함수를 시작할 때 항상 똑같이 수행하는 명령으로 프롤로그(Prologue)라고 부름

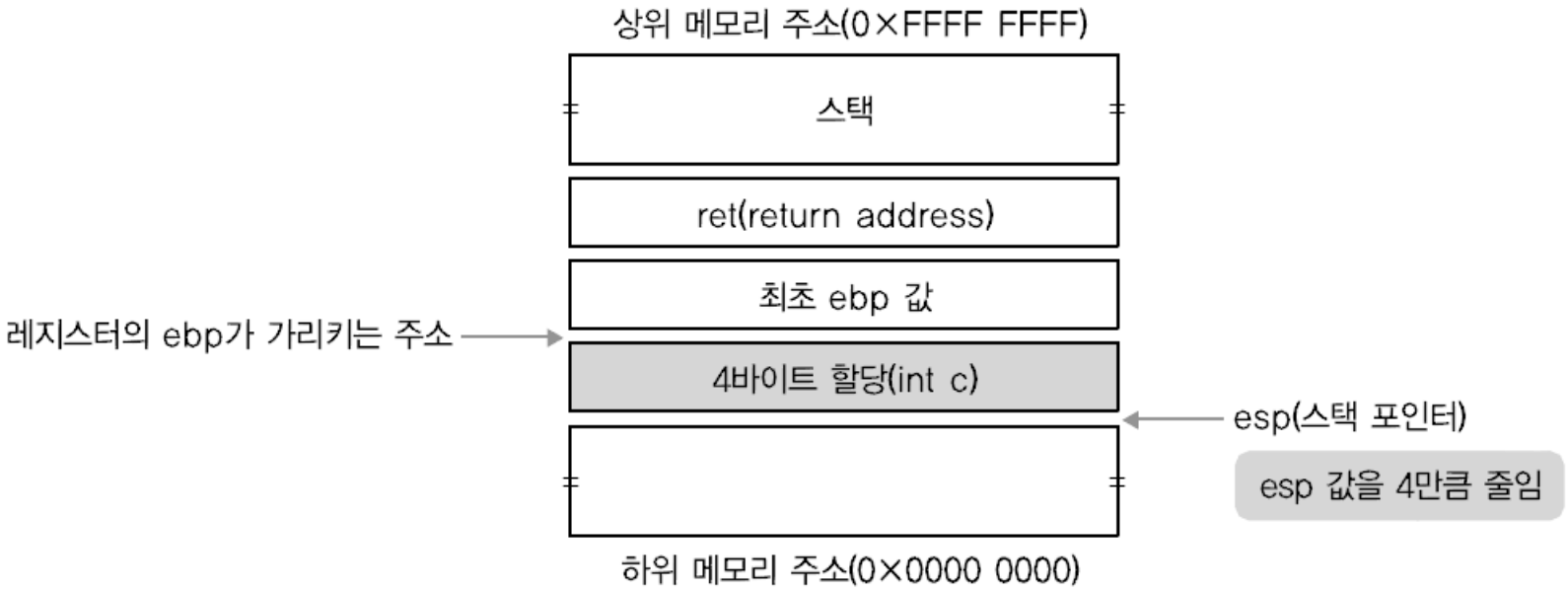


[그림 2-28] movl %esp, %ebp 실행 시 스택의 구조

기본 코드 분석

❸ subl $4, %esp

– 스택에 4바이트만큼의 용량을 할당



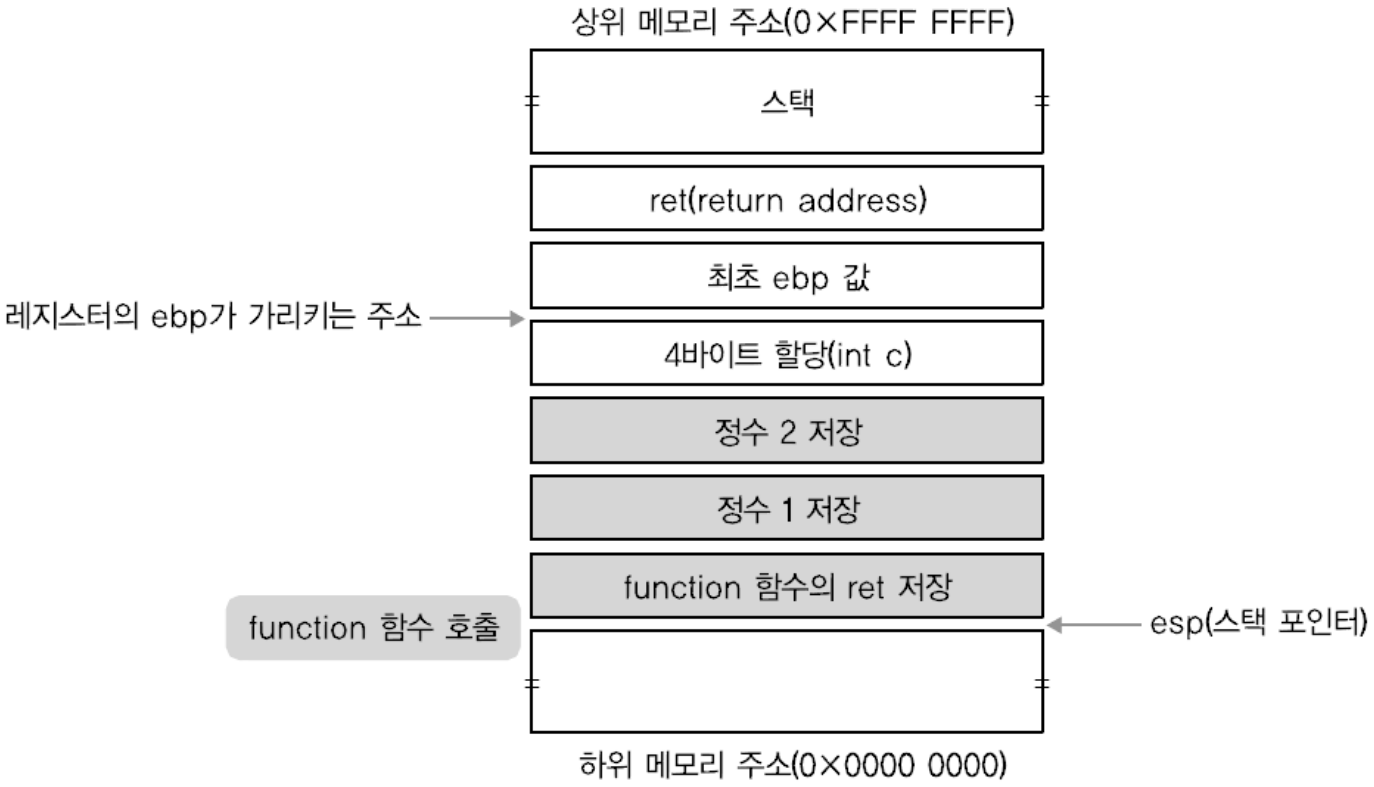
[그림 2-29] subl $4, %esp 실행 시 스택의 구조

기본 코드 분석

❹ pushl $2 : ❹ ~ ❻ 세 단계는 function(1, 2)에 대한 코드임

❺ pushl $1

❻ call function

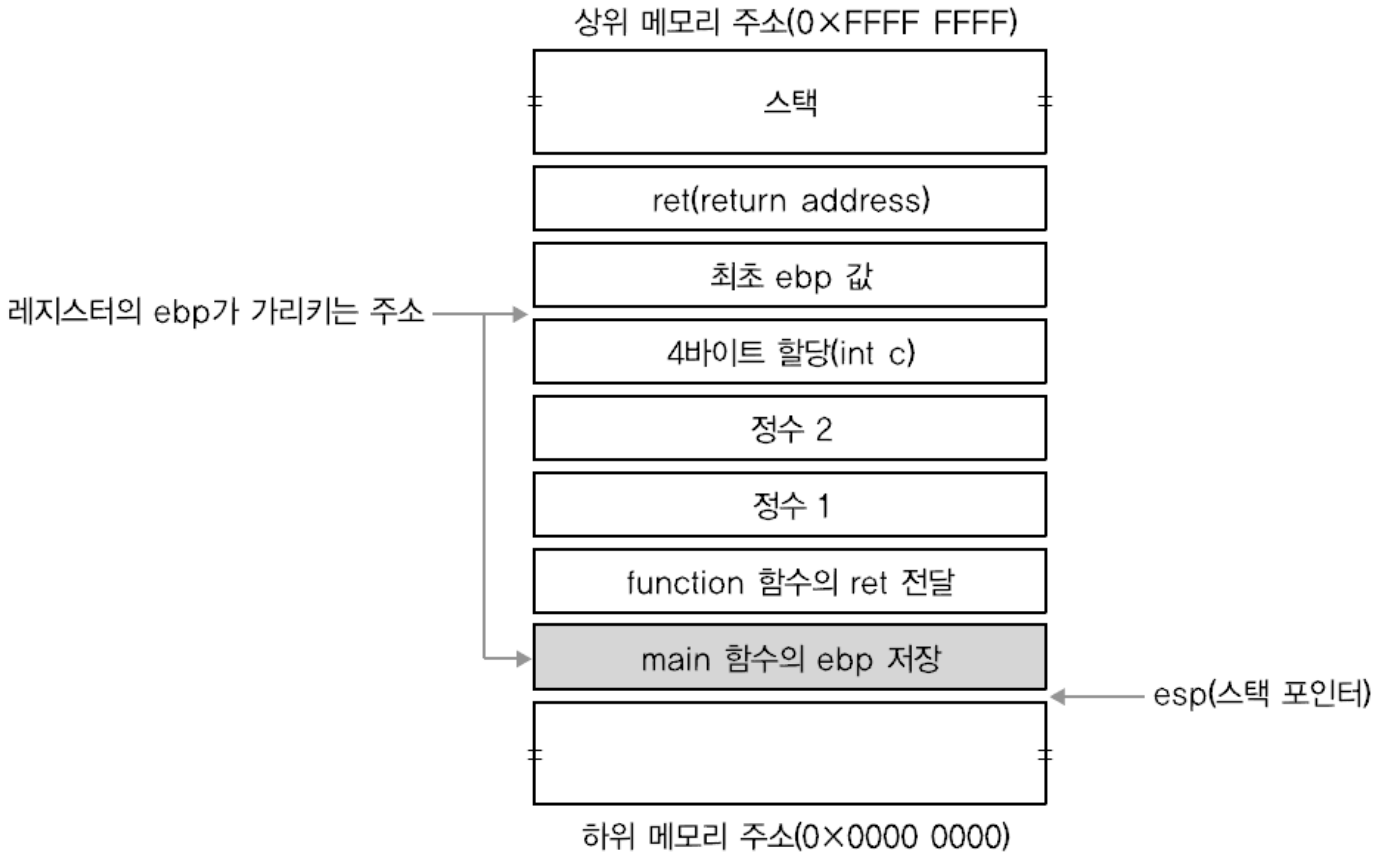


[그림 2-30] pushl $2, pushl $1, call function 실행 시 스택의 구조

기본 코드 분석

❼ pushl %ebp

– 현재 레지스터의 ebp 값을 스택에 저장

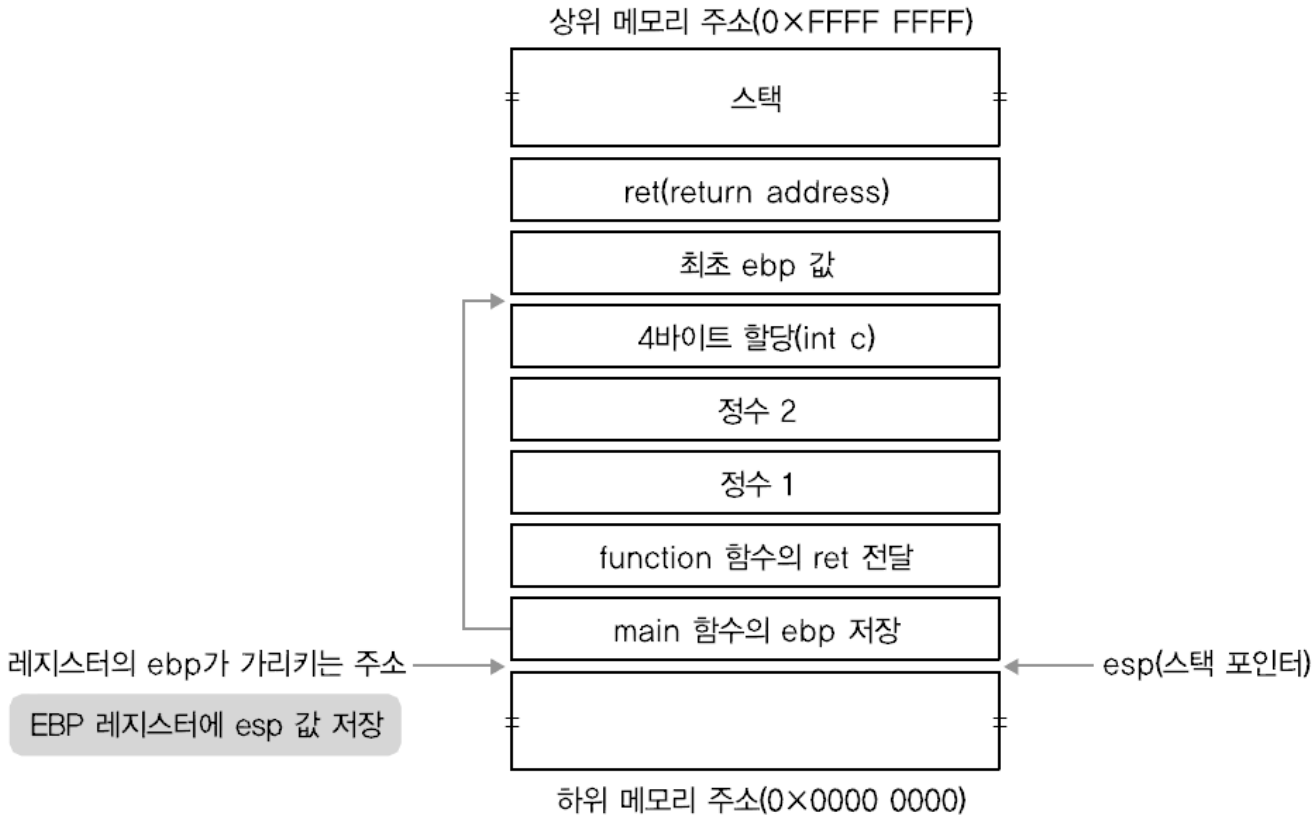


[그림 2-31] pushl %ebp 실행 시 스택의 구조

기본 코드 분석

❽ movl %esp,%ebp

– function(1, 2)의 시작에서도 pushl %ebp 명령과 movl %esp,%ebp이 실행됨

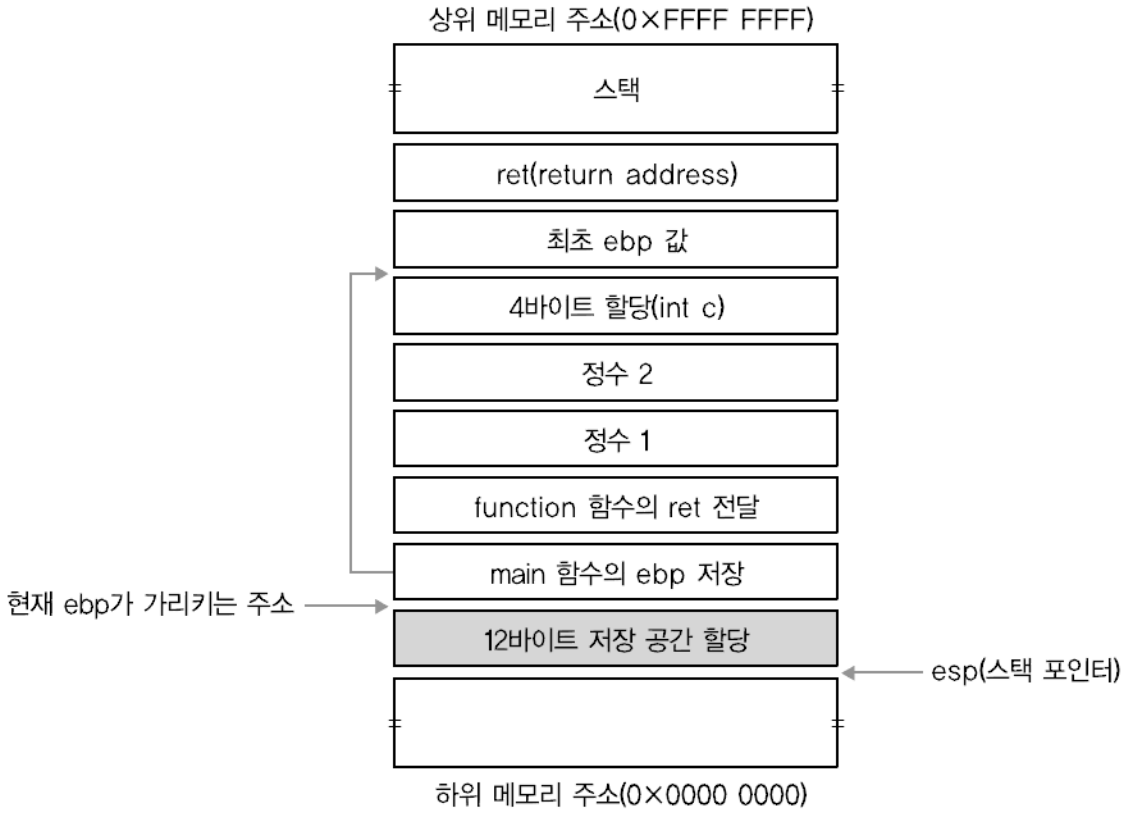


[그림 2-32] movl %esp,%ebp 실행 시 스택의 구조

기본 코드 분석

❾ subl $12,%esp

- char buffer[10] 할당
- 스택에 12바이트만큼 용량을 할당

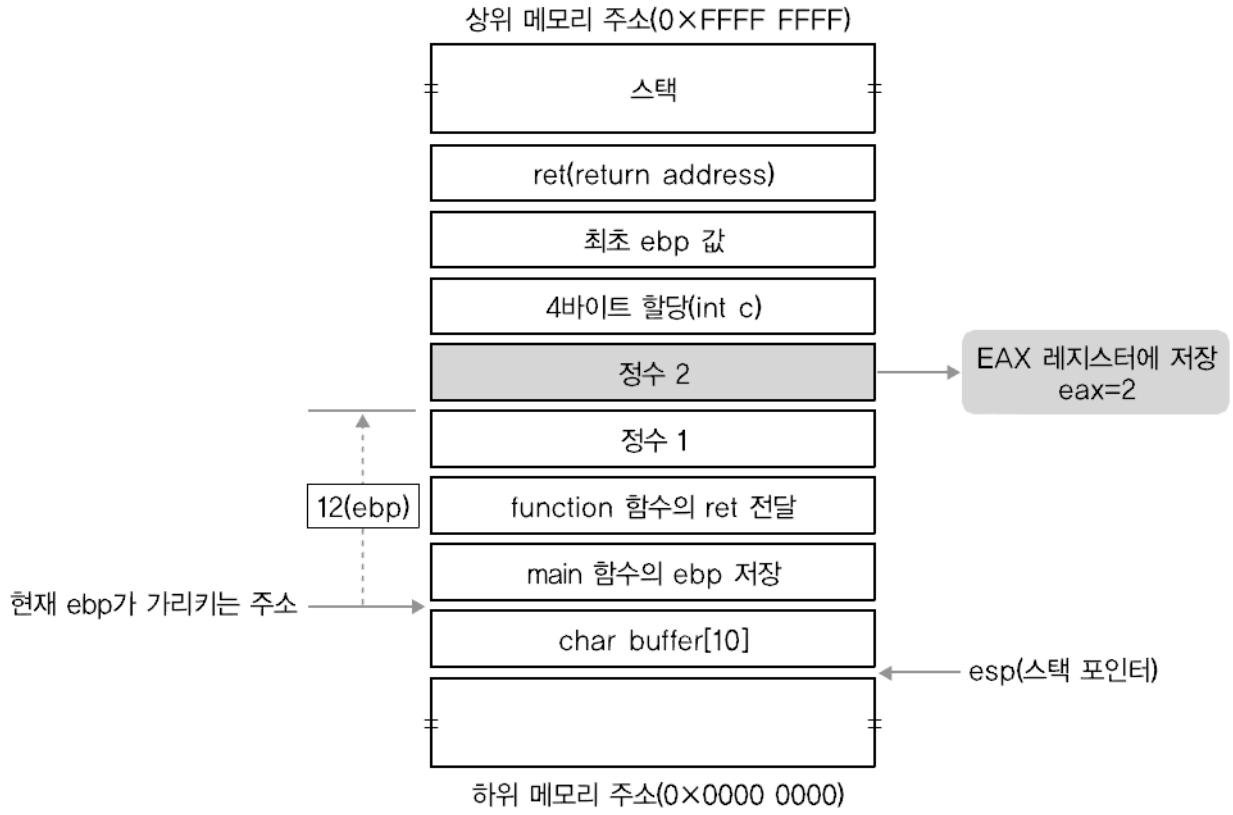


[그림 2-33] subl $12,%esp 실행 시 스택의 구조

기본 코드 분석

⑩ movl 12(%ebp),%eax

– ebp에 12바이트를 더한 주소 값의 내용(정수 2)을 eax 값에 복사

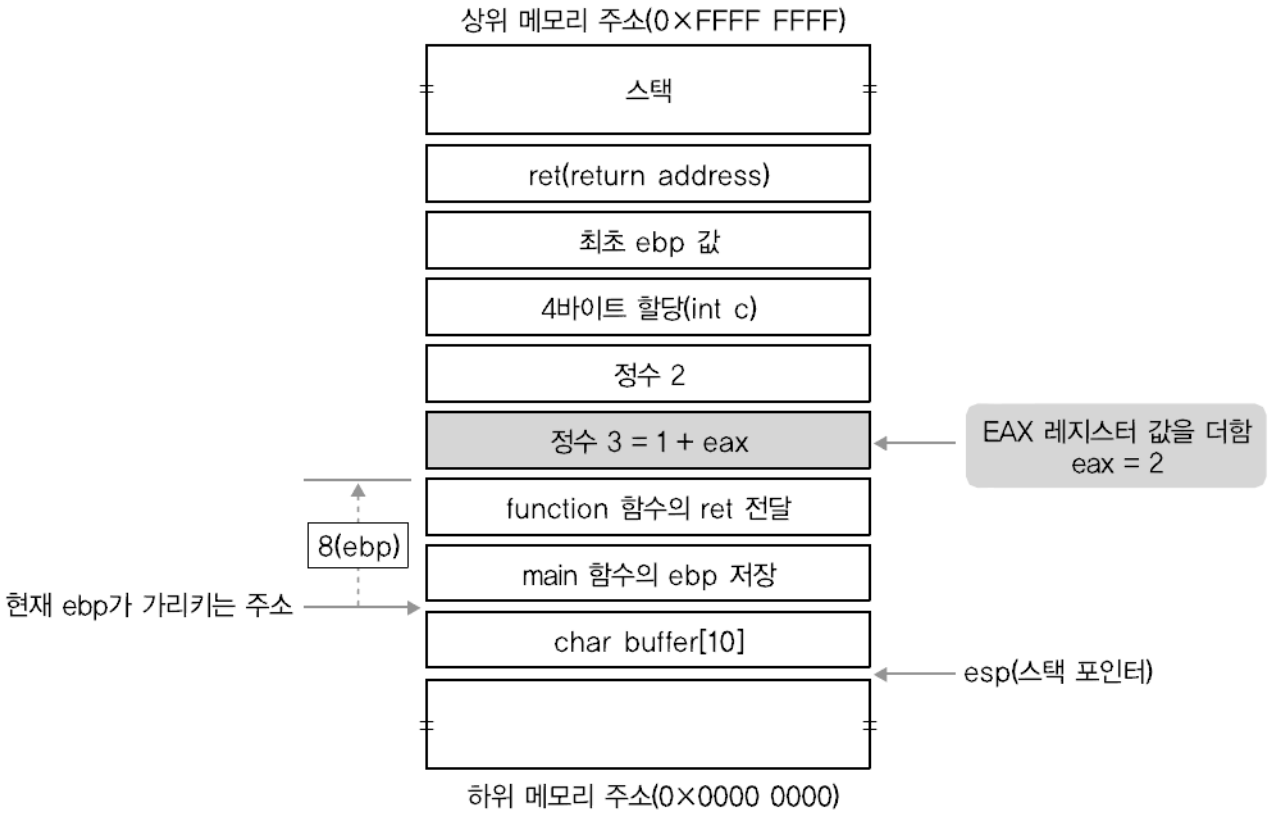


[그림 2-34] movl 12(%ebp),%eax 실행 시 스택의 구조

기본 코드 분석

⓫ addl %eax,8(%ebp)

– ebp에 8바이트를 더한 주소 값의 내용(정수 1)에 eax(단계 10에서 2로 저장됨) 값을 더함

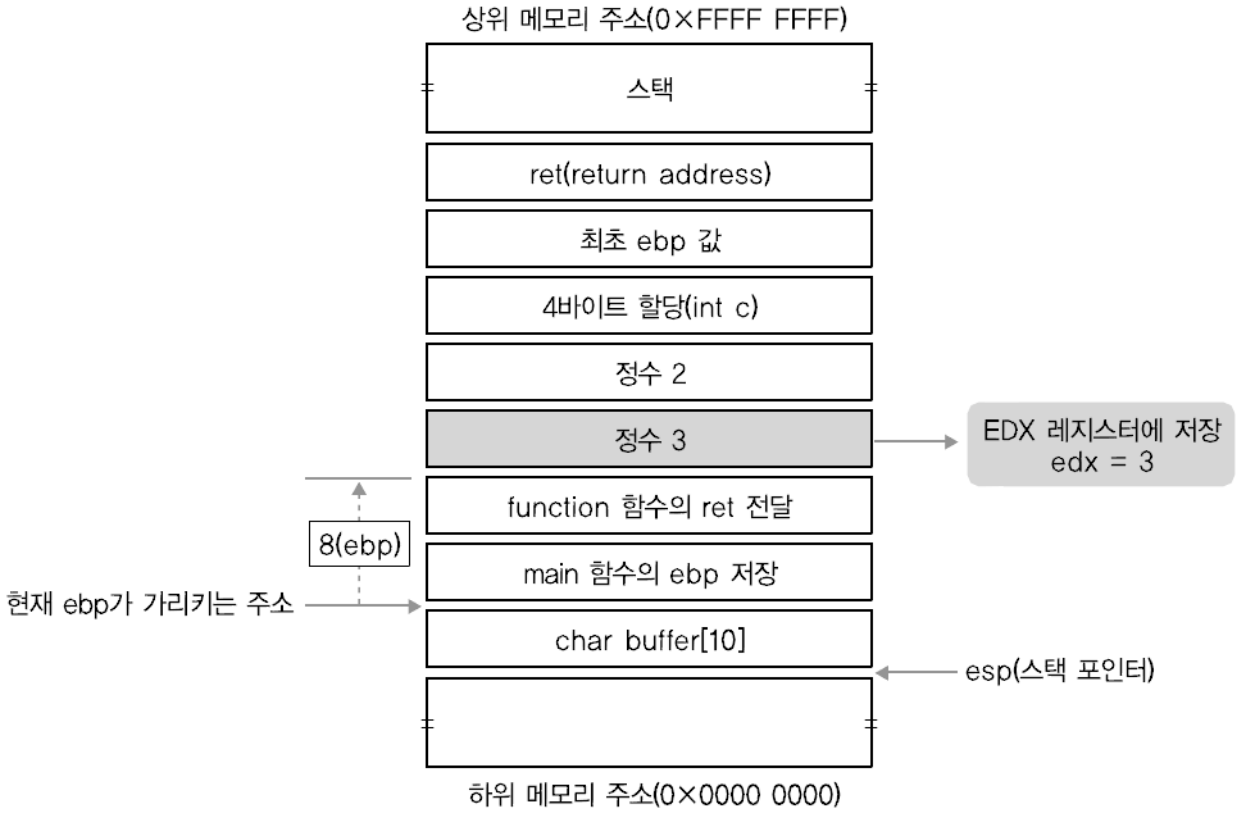


[그림 2-35] addl %eax,8(%ebp) 실행 시 스택의 구조

기본 코드 분석

⓬ movl 8(%ebp),%edx

– ebp에 8바이트 더한 주소 값의 내용(정수3)을 edx에 저장



[그림 2-36] movl 8(%ebp),%edx 실행 시 스택의 구조

기본 코드 분석

⓭ movl %edx,%eax

- edx에 저장된 정수 3을 eax로 복사

⓮ jmp .L1

- L1로 점프

⓯ leave

- 함수를 끝냄

⓰ ret

- function 함수를 마치고 function 함수에서 저장된 ebp 값을 제거
- main 함수의 원래 ebp 값으로 EBP 레지스터 값 변경

기본 코드 분석

⓱ addl $8,%esp

- esp에 8바이트를 더함

⓲ movl %eax,%eax

- eax 값을 eax로 복사

⓳ movl %eax,-4(%ebp)

- ebp에서4바이트를 뺀 주소 값(int c)에 eax 값을 복사

⓴ leave

■ ret

THANK YOU!