# 어셈블리 튜토리얼 1

HandyPost는 한 도영(HDNua)이 작성하는 포스트 문서입니다.

## 1. 개요

어셈블리를 배우기 위한 중간 단계 언어인 Handy CIL 언어의 개념과 활용을 학습함으로써 컴파일러 구현을 위한 준비를 한다.

## 2. 프로젝트 준비

이전에 사용하던 예제에서는 C++ 프로그래밍 언어를 사용하였다. 이 문서에서는 C++ 프로그래밍 언어가 아닌 C 언어를 이용하여 프로젝트를 작성하는데, 왜냐하면 C++는 C가 아니며, C에서 되던 것이 C++에서 제한되는 경우가 있기 때문이다. 후에 다시 얘기하지만 C++ 프로젝트로 진행하면 컴파일 오류가 발생하는 예제가 포함되어있다. 혹 아래에 제시하는 예제들을 직접 실행해보고 싶다면 소스 파일 등의 자료가 올라오는 github 페이지(https://github.com/HDNua/JSCC)에서 예제 프로젝트를 내려 받거나, 필자의 블로그(http://blog.naver.com/rbfwmqwntm)에서 이를 참조하라.

시스템은 32bit 운영체제를 기본으로 한다. 이는 모든 기본 변수의 크기가 4바이트로 고정되어있음을 의미한다. Windows에서 Visual Studio 2013을 이용해 프로그래밍 하는 경우에는 코드의 변경 없이 빌드 할 수 있으나, 다른 운영체제 또는 다른 도구를 사용할 때는 해당 시스템에 맞게 CIL 헤더 파일의 정의를 다음과 같이 변경해야 할 수 있다.

```
#define SYSTEM_BIT 64 // 32
```

참고로 이 문서는 꼭 컴파일러를 만들기 위한 목적이 아닌, 순수하게 어셈블리 언어를 배우려는 목적으로도 활용할 수 있다. 즉, 어셈블리 언어를 공부하기 위해서라면 이전 문서를 볼 필요는 없다.

## 3. 중간 단계 언어를 이용한 프로그래밍

이전 문서에서 기계어보다는 사용하기 쉽고, 고급 프로그래밍 언어보다는 기계가 이해하기 쉬운 중간 단계 언어가 필요하다는 사실을 알았다. 그런데 중간 단계 언어가 필요하다는 사실은 알았지만 아직 느낌이 크게 없다. 중간 단계 언어라는 것을 이용하여 직접 프로그램을 작성할 수 있을 정도면 좋겠다는 생각이 들지 않는가? 이를 위해 필자는 중간 단계 언어를 체험할 수 있도록 C 프로그래밍 언어를 이용하는 방법을 생각해보았다. 예를 들면 위에서도 말했듯 C 프로그래밍 언어에서 변수를 마음대로 선언하는 것은 사실 복잡한 과정이고, 실제 내부에서는 몇 개의 변수만 사용하여 메모리를 조작하는데, 그렇다면 C 프로그래밍 언어로 프로그램을 작성할 때 변수를 임의로 생성하지 말고 주어진 변수만으로 해결한다면 중간 단계 언어를 체험할 수 있을 것이라는 생각이 든 것이다. 직접 해보자.

### 3.1) Handy CIL 프로그래밍 언어

이 문서에서 사용할 중간 단계 언어를 Handy C Intermediate Language라고 하겠다. 이 문서에서는 이를 CIL이라고 부를 것이다. 다음은 중간 단계 언어로 작성한 HelloWorld 프로그램이다. 다시 말하지만 C를 이용하여 중간 단계 언어를 흉내 낸 것이 불과하다.

HelloWorld.c

```
#include "CIL.h"

STRING sHelloWorld = "Hello, world!"; // 프로그램에 사용할 문자열을 정의합니다.

PROC(main) // main 프로시저의 시작 지점입니다. PROC은 procedure의 줄임말입니다.
```

```
PUSH(sHelloWorld) // 콘솔에 문자열을 출력하기 위해 인자를 저장합니다.
INVOKE(print_str) // 콘솔에 문자열을 출력하는 프로시저를 호출합니다.

ENDP // 프로시저의 정의가 종료되는 지점입니다.
```

한 줄 한 줄 분석해보자.

- CIL 헤더 파일: 중간 단계 언어를 체험하기 위해 필자가 작성한 매크로가 정의된 파일이다.
- STRING: const char * 형식으로 정의되어있으며, C 형식의 문자열과 정의하는 방법이 같다.
- PROC(main): main 함수의 시작 지점이다. 함수는 프로시저(procedure)라고도 하는데, 앞으로 CIL에서는 함수라는 용어 대신 프로시저(procedure)라는 용어를 사용할 것이다.
- PUSH(sHelloWorld): CIL 프로그래밍 언어는 중간 단계이므로 함수를 호출하는 데 제약이 있다. C에서는 코드 print_str("Hello, world!")를 작성하면 문자열이 잘 출력되지만, CIL에서는 다음과 같이 세 단계로 분리해야 한다.

> 프로시저 시작 이전에 문자열 sHelloWorld를 ("Hello, world!")로 정의한다.
> print_str 프로시저를 호출하기 전에 sHelloWorld를 PUSH 명령을 이용해 인자로 보관한다.
> INVOKE 매크로를 이용해 print_str 프로시저를 호출한다. 결과로 문자열이 출력된다.

- INVOKE(print_str): 방금 말했듯, print_str 프로시저를 호출한다.
- ENDP: 프로시저의 끝을 나타낸다. main 프로시저가 끝나면 프로그램이 종료된다.

예제는 아주 단순하다. 이해하는 데 무리가 없으리라 생각한다. 주의할 점이라면, CIL의 문장의 마지막에는 세미콜론(;)을 사용하지 않아도 되며 한 줄에 하나의 명령만 가능하다. 다만 PROC, ENDP와 같은 키워드에는 세미콜론을 사용하면 안 되는데 이에 대해서는 후에 자세히 다루겠다.

**3.2) 정수와 문자열을 출력하기**

CIL 프로그래밍 언어에서는 정수를 출력하는 프로시저 print_int를 지원한다. 다음은 이를 이용하여 식을 계산하고 그 결과를 출력하는 프로그램의 소스 코드이다.

PrintValue.c

```
#include "CIL.h"

STRING sNewLine = "\n"; // 개행 문자를 삽입하기 위한 개행 문자열입니다.
STRING sResult = "RESULT: ";

PROC(main) // main 프로시저의 정의가 시작되는 지점입니다.

// 정수 0을 출력합니다.
PUSH(0)
INVOKE(print_int)

// 출력 화면에 개행 문자를 삽입합니다.
PUSH(sNewLine)
INVOKE(print_str)

// 문자열 "RESULT: "를 출력합니다. (개행 문자가 포함되지 않습니다!)
PUSH(sResult)
INVOKE(print_str)
```

```
// 준비된 변수 a에 정수를 대입합니다.
MOVL(a, 100)

// a에 저장된 정수를 출력합니다.
PUSH(a)
INVOKE(print_int)

// 출력 화면에 개행 문자를 삽입합니다.
PUSH(sNewLine)
INVOKE(print_str)

ENDP // 프로시저의 정의를 마칩니다.
```

실행 결과

```
0
RESULT: 100
```

주석이 잘 되어있어 이해하는 데 무리가 없으리라 생각하지만, 결과가 어떻게 나왔는지 그 과정을 이해하는 것이 매우 중요하다.

**3.3) 기본으로 제공되는 변수를 이용해 연산하기**

여기서 하나 더 중요한 사실을 말하자면, CIL 언어에서는 기본적으로 사용할 수 있는 변수의 수가 제한되어있다. 이 문서에서는 이러한 변수를 **기본 변수**라고 하겠다. 다음 예제에서는 이들을 이용하여 연산을 해보고 그 결과를 출력하는 프로그램을 작성해본다.

Calculation.c

```
#include "CIL.h"
STRING sNewLine = "\n"; // 개행 문자를 삽입하기 위한 개행 문자열입니다.

PROC(main) // main 프로시저의 정의가 시작되는 지점입니다.

// 준비된 변수 a에 10을 대입하고 a를 출력합니다.
MOVL(a, 10) // a = 10

// a를 출력합니다.
PUSH(a)
INVOKE(print_int)

// 개행 문자를 삽입합니다.
PUSH(sNewLine)
INVOKE(print_str)

// 변수 a에 10을 더하고 a를 출력합니다.
ADD(a, 10) // a = 10
```

```
// a를 출력합니다.
PUSH(a)
INVOKE(print_int)

// 개행 문자를 삽입합니다.
PUSH(sNewLine)
INVOKE(print_str)

// 준비된 변수 b에 14를 대입하고 a에서 b를 뺍니다.
MOVL(b, 14) // b = 14
SUB(a, b) // a -= b

// a를 출력합니다.
PUSH(a)
INVOKE(print_int)

ENDP // 프로시저의 정의를 마칩니다.
```

| 실행 결과 |
|---|
| 10<br>20<br>6 |

CIL은 추가적인 C 변수 선언을 허용하지 않는다. 다음은 이미 정의된 C 변수 중 일부이다.

- a: accumulator. 모든 연산의 결과가 누적되는 누산 변수다.
- c: counter. 반복문에서 반복 횟수를 결정할 때 참조하는 카운터 변수다.
- d: data. 연산에서 임시로 사용하는 데이터를 보관하는 변수다.
- b: base. 여기서는 거의 사용하지 않을 변수로 봐도 좋다.

CIL은 C보다 저급 언어이므로 복합 연산을 지원하지 않기 때문에, 한 번에 하나씩의 연산만 수행해야 한다. 1+2*3+4와 같은 식을 예로 들면, CIL에서는 이 식을 다음과 같이 표현한다.

```
MOVL(a, 1) // a = 1
MOVL(b, 2) // b = 2
MUL(b, 3) // b *= 3
ADD(a, b) // a += b
ADD(a, 4) // a += 4
```

### 3.4) 프로그램 흐름 제어

다음은 CIL의 코드 제어 명령 중 하나인 점프문이다.

Jump.c

```
#include "CIL.h"
STRING sHello = "HelloWorld\n";
STRING sNice = "NiceToMeetYou\n";
```

```
STRING sBye = "GoodBye\n";
PROC(main) // main 프로시저의 정의가 시작되는 지점입니다.

PUSH(sHello)
INVOKE(print_str)

JMP(label) // label 레이블로 점프합니다.

PUSH(sNice)
INVOKE(print_str)

label: // label 레이블의 정의입니다.

PUSH(sBye)
INVOKE(print_str)

ENDP // 프로시저의 정의를 마칩니다.
```

| 실행 결과 |
| --- |
| HelloWorld<br>GoodBye |

프로그램을 실행하면 JMP 문장이 실행된 지점과 label 레이블이 정의된 문장 사이의 명령이 모두 생략되었음을 알 수 있다. 이렇듯 JMP 문장은 C의 goto 명령과 같다(실제로 헤더에 그렇게 정의되어있다).

이제 CIL의 조건문을 보자.

Condition.c

```
#include "CIL.h"
PROC(main) // main 프로시저의 정의가 시작되는 지점입니다.

// a = 10, b = 20
MOVL(a, 10)
MOVL(b, 20)

// a와 b를 서로 비교하고 결과를 flag에 저장합니다.
CMP(a, b)

// a와 b의 차이가 0이 아니라면 elseif 레이블로 점프합니다.
JNZ(lbl_elseif)

// elseif 레이블로 점프하지 않으면 a = 30을 수행한 후
MOVL(a, 30)
// endif 레이블로 점프합니다.
```

```
JMP(lbl_endif)

// elseif 레이블로 점프했다면
lbl_elseif:
// a = 40을 수행합니다.
MOVL(a, 40);

lbl_endif:
PUSH(a)
INVOKE(print_int)

ENDP // 프로시저의 정의를 마칩니다.
```

CIL은 조건 분기할 때 플래그 변수를 참조한다. CMP 명령은 인자로 넘어온 두 값을 비교해서 결과를 플래그 변수에 저장하는데, 이때 저장하는 정보는 두 값이 서로 같은지, 왼쪽이 더 큰지(부호)와 같은 것들이다. 이 예제에서는 a와 b의 값이 서로 다르므로 lbl_elseif 레이블로 이동하고 a에는 40이 저장된다.

다음은 조건문과 점프문을 이용하여 구성한 반복문이다.

Loop.c

```
#include "../CIL/CIL.h"
STRING sNewLine = "\n";
STRING sEnd = "Program end\n";

PROC(main) // main 프로시저의 정의가 시작되는 지점입니다.

MOVL(c, 5); // c = 5

// 루프의 시작을 뜻하는 레이블을 정의합니다.
loop_start:

// c의 값을 0과 비교합니다.
CMP(c, 0);

// c가 0이라면 반복문을 탈출합니다.
JZ(loop_end)

// c의 현재 값을 출력합니다.
PUSH(c);
INVOKE(print_int);
PUSH(sNewLine);
INVOKE(print_str);

// c--
DEC(c);
```

```
// 루프의 처음으로 되돌아가 반복문을 다시 실행합니다.
JMP(loop_start);

// 루프의 끝을 뜻하는 레이블을 정의합니다.
loop_end:

PUSH(sEnd);
INVOKE(print_str);

ENDP // 프로시저의 정의를 마칩니다.
```

| 실행 결과 |
|---|
| 5<br>4<br>3<br>2<br>1<br>Program end |

위 프로그램은 카운터 변수 C를 이용하여 5부터 0이 아닐 때까지 반복하여 수를 출력한다. 이 예제를 이해한다면 C를 배울 때 연습하던 별 찍기와 같은 문제들도 모두 CIL을 이용해 해결할 수 있다. 심심할 때 연습 삼아 풀어보면 재미있을 것이다.

**3.5) 프로시저(procedure)**

잘 알고 있듯이 main 프로시저의 본체에만 코드를 작성하면 가독성과 생산성이 아주 나빠지므로, 가능한 한 프로시저를 역할 별로 분리하는 것이 좋다. CIL에서도 여러 개의 프로시저를 사용자가 정의하고 호출할 수 있다.

Procedure.c

```
#include "../CIL/CIL.h"
STRING sHello = "Hello, procedure!";

// 프로시저 hello를 정의합니다.
PROC(hello)

PUSH(sHello);
INVOKE(print_str)

ENDP

// main
PROC(main)
```

```
// hello 프로시저를 호출합니다.
INVOKE(hello)

ENDP
```

여기에는 흥미로운 사실 하나가 더 있는데, 같은 파일 내부라면 순서를 생각할 필요 없이 프로시저를 정의하고 호출할 수 있다는 것이다. 즉 다음은 적법한 코드이다.

Procedure.c

```
#include "CIL.h"
STRING sHello = "Hello, procedure!";

// main
PROC(main)
// hello 프로시저를 호출합니다.
INVOKE(hello)
ENDP

// 프로시저 hello를 정의합니다.
PROC(hello)

PUSH(sHello);
INVOKE(print_str)

ENDP
```

정상적인 C 프로젝트를 만들었음에도 불구하고 이것이 지원되지 않는다면, 컴파일러가 내부적으로 이를 지원하지 않을 가능성이 있다. 이 경우 호출 전, 즉 코드의 위 부분에 다음과 같이 적는다.

`PROTO(<procedure_name>); // ex) PROTO(hello);`

참고로 이 경우 컴파일러가 오류를 뱉는 건, 컴파일러가 최신 표준을 잘 지키고 있기 때문이다. 구형 C에서는 암묵적인 함수 호출이 표준에 있었지만, C99 표준 이후로는 모두 금지되어있다. Visual Studio 2013은 이것이 가능한 개발 도구 중의 하나인데, 이유는 하위 호환 때문이니 결코 Visual Studio가 다른 컴파일러보다 우수하거나, 다른 컴파일러가 뒤떨어지는 것으로 오해하는 일이 없었으면 좋겠다.

### 3.6) 메모리

기본 변수 m을 이용해 메모리에 직접 접근할 수 있다.

Memory.c

```
#include "CIL.h"
STRING sNewLine = "\n";

// main
PROC(main)

// 기본 변수 a를 8로 초기화합니다.
```

```
MOVL(a, 8) // a = 8

// 정수를 출력하고 개행합니다.
PUSH(a)
INVOKE(print_int)
PUSH(sNewLine)
INVOKE(print_str)

// 메모리 10번지에 값을 설정합니다.
// *(int *)(m[10]) = 20;
SETL(m + 10, 20);

// 메모리 10번지에서 값을 획득하여 a에 저장합니다.
// a = *(int *)(m[10]);
GETL(a, m + 10);

// 획득한 값을 출력하여 올바른지 확인합니다.
PUSH(a)
INVOKE(print_int)

ENDP
```

CIL은 메모리에 직접 접근이 가능한 저급 언어이므로, 위와 같이 메모리에 직접적으로 값을 쓸 수 있다. C에서도 마찬가지지만 이러한 행위는 아주 위험하기 때문에, 메모리를 사용할 때는 운영체제가 제공하는 함수를 사용하거나, 프로시저 내에 지역 변수를 만드는 것이 일반적이다.

**3.7) 지역 변수**

CIL이 추가적인 C 변수 선언을 허용하지 않는다고 하였다. 하지만 우리는 프로시저 내에 지역 변수를 만들어 사용할 수 있는데, 그 방법은 다음과 같다.

- 지역 변수로 사용할 공간을 확보한다.
- 확보한 공간의 주소를 기억해놓고, 필요할 때마다 해당 주소에 접근한다.

다음은 이를 구현하는 코드이다. 이전에 사용하지 않던 기본 변수를 사용하므로 주의 깊게 봐야 한다.

LocalVariable.c

```
#include "CIL.h"
STRING sNewLine = "\n";

// main
PROC(main)

// sp는 현재 메모리의 위치를 표시하는 기본 변수입니다.
// sp의 값을 정수로 출력하고 개행합니다.
PUSH(sp)
INVOKE(print_int)
PUSH(sNewLine)
```

```
INVOKE(print_str)

// sp 기본 변수의 값을 4만큼 뺍니다.
// sp -= 4;
// 4byte는 32bit 정수형 변수의 크기입니다.
SUB(sp, 4)

// sp가 가리키는 메모리의 주소 값을 a에 복사합니다.
// a = &m[sp];
// CIL은 자료형에 엄격하지 않습니다.
LEA(a, m + sp)

// a의 값을 주소 값으로 간주하고 해당 주소에 값을 설정합니다.
// *a = 10;
SETL(a, 10)

// sp가 가리키는 메모리의 값을 획득하여 a에 복사합니다.
GETL(a, m + sp)

// a의 값을 출력합니다.
PUSH(a)
INVOKE(print_int)

ENDP
```

값을 출력하는 코드를 제외하면 사실 4줄밖에 안 되는 단순한 코드다. 다만 여기에서 메모리를 좀 더 명확하게 그림으로 나타내어야 이후에 이야기를 진행할 때 문제가 없을 것 같다.

이미 알고 있겠지만, 메모리는 바이트의 배열이다. CIL에서는 메모리에 접근하기 위한 기본 변수를 제공하는데, 이 변수에 대해 먼저 정리하자.

- m: memory. 시스템의 메모리를 나타내는 배열 변수다.
- sp: stack pointer. 프로그램 실행 시에 생성되는 스택 메모리를 가리키는 포인터 변수다.
- bp: base pointer. 프로시저 호출 시에 스택의 시작 주소를 저장하는 포인터 변수다.

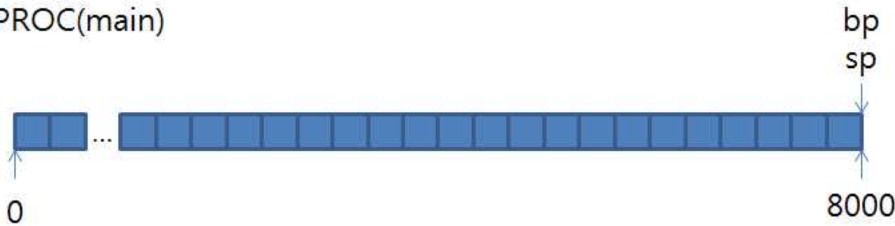
bp에 대해서는 좀 더 나중에 다루고, 일단 m과 sp만을 얘기해보자. 스택이라고 하니 1장에서 배웠던 스택을 떠올릴 수 있는데, 맞는 판단이고 실제로 스택 형태로 메모리가 관리되지만, 여기서는 일단 스택 영역이라는 메모리 배열이 선언되어있고 sp가 이 배열을 가리키는 형태라고 생각하는 게 이해하기 편할 것 같다(물론 후에 왜 이것이 스택인지를 설명할 것이다). 독자의 편의를 위해 위에 제시한 코드에서 값을 출력하는 등의 쓸모없는 부분을 제외한 코드를 보이겠다.

LocalVariable.c

```
#include "CIL.h"
PROC(main)
SUB(sp, 4)
LEA(a, m + sp)
SETL(a, 10)
GETL(a, m + sp)
```

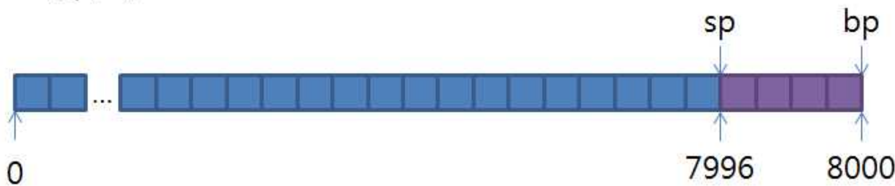
```
ENDP
```

프로시저가 호출되면 메모리는 다음 상태가 된다.



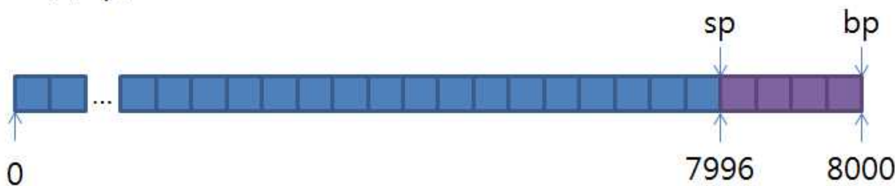

| 기본 변수 | 값 |
|---|---|
| a | ? |
| sp | 8000 |
| bp | 8000 |

이제 스택 포인터에서 값을 빼는 SUB 연산을 수행한다.



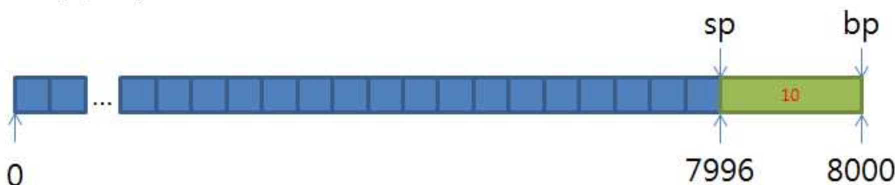

| 기본 변수 | 값 |
|---|---|
| a | ? |
| sp | 7996 |
| bp | 8000 |

그리고 sp가 가리키는 메모리의 주소를 LEA(load effective address) 명령으로 획득한다.



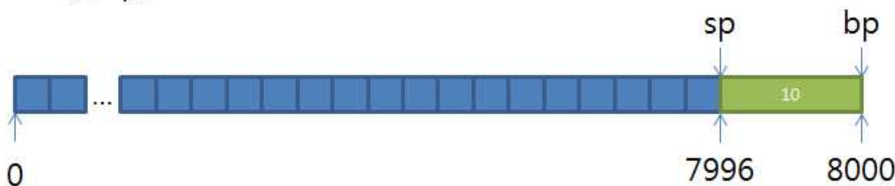

| 기본 변수 | 값 |
|---|---|
| a | 7996 |
| sp | 7996 |
| bp | 8000 |

SETL 명령을 이용하여 획득한 주소에 10을 저장한다.




| 기본 변수 | 값 |
|---|---|
| a | 7996 |
| sp | 7996 |
| bp | 8000 |

마지막으로 GETL 명령을 이용하여 기본 변수 a에, sp가 가리키는 메모리의 값을 획득하여 저장한다.

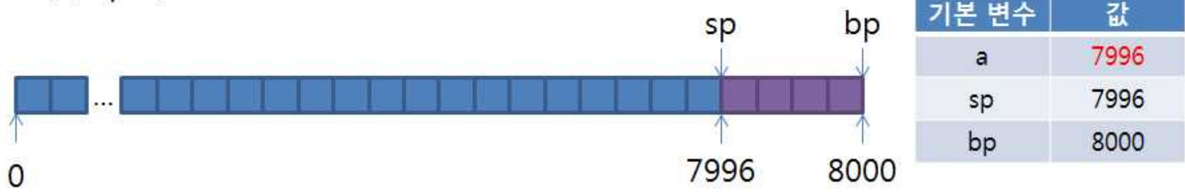



| 기본 변수 | 값 |
|---|---|
| a | 10 |
| sp | 7996 |
| bp | 8000 |

지역 변수를 두 개 이상 사용해야 하는 경우에는 스택의 시작 주소를 기준으로 변수의 크기만큼을 뺀다. 여기서 용어를 하나 정의하자. 어떤 두 대상의 수치적 거리를 **오프셋(offset)**이라고 한다. 따라서 위 그림에서 기본 변수 sp와 bp의 오프셋은 4바이트가 된다. 이는 아주 중요한 내용인데, 이 예제에서는 만든 지역 변수에 접근하기 위해 LEA 명령을 이용할 때 sp 기본 변수를 이용했지만, 일반적으로는 기본 변수 bp와 해당 지역 변수의 오프셋의 합을 이용한다. 예를 들어 이 예제에서는 다음이 성립한다.

만든 지역 변수의 위치 `== sp == (bp - 4)`

따라서 LEA 명령을 다음과 같이 수행해도 문제되지 않는다.




| 기본 변수 | 값 |
|---|---|
| a | 7996 |
| sp | 7996 |
| bp | 8000 |

이것이 왜 중요하냐면, 지역 변수의 위치를 계산할 때는 반드시 bp를 이용해야 하기 때문이다. 지역 변수를 두 개 만드는 상황을 가정하자.

LocVarSp.c

```
#include "CIL.h"
STRING sNewLine = "\n";

// main
PROC(main)

// 4바이트 지역 변수를 생성합니다. 임시로 var1이라고 합시다.
// int var1;
// 현재 var1과 sp의 오프셋은 0byte입니다.
SUB(sp, 4)

// var1 = 10;
// sp는 현재 var1을 가리킵니다.
SETL(m + sp, 10);

// 4바이트 지역 변수를 더 생성합니다. 임시로 var2라고 합시다.
// int var2;
// 현재 var2와 sp의 오프셋은 0byte입니다.
// 현재 var1과 sp의 오프셋은 4byte입니다.
SUB(sp, 4)

// var2 = 20;
// sp는 현재 var2를 가리킵니다.
SETL(m + sp, 20);

// sp를 기준으로 var1과 var2의 값을 획득합니다.
GETL(a, m + sp); // (sp)가 가리키는 값을 획득합니다.
GETL(b, m + sp + 4); // (sp+4)가 가리키는 값을 획득합니다.

ENDP
```

이 코드의 문제점이 무엇인지 알겠는가? 바로 변수가 새롭게 생성됨에 따라, 같은 변수를 가리키는 데 sp와의 오프셋이 계속 달라진다는 점이다. C는 변수 선언이 함수의 앞에 모두 위치하므로 크게 문제되지 않는다고 할지 모르나, sp는 지역 변수를 선언하는 데만 사용되는 변수가 아니다. 프로시저 호출

을 위한 인자를 전달할 때, 프로시저를 호출할 때도 sp를 사용한다. 즉 sp를 이용하여 지역 변수에 접근하려면 프로시저 내에서 언제 그 값이 변하는지를 몽땅 추적해서 그 오프셋을 이용해야 한다.

이 문제는 스택의 시작 주소를 보관하는 bp 기본 변수를 이용하면 해결할 수 있다. 다음은 위 코드를 bp 기본 변수를 이용하여 변경한 것이다.

LocVarBp.c

```
#include "CIL.h"
STRING sNewLine = "\n";

// main
PROC(main)

// 4바이트 지역 변수를 생성합니다. 임시로 var1이라고 합시다.
// int var1;
// 현재 var1과 bp의 오프셋은 4byte입니다.
SUB(sp, 4)

// var1 = 10;
// (bp-4)는 var1을 가리킵니다.
SETL(m + bp - 4, 10);

// 4바이트 지역 변수를 더 생성합니다. 임시로 var2라고 합시다.
// int var2;
// 현재 var2와 bp의 오프셋은 8byte입니다.
// 현재 var1과 bp의 오프셋은 4byte입니다.
SUB(sp, 4)

// var2 = 20;
// (bp-8)은 var2를 가리킵니다.
SETL(m + bp - 8, 20);

// sp를 기준으로 var1과 var2의 값을 획득합니다.
GETL(a, m + bp - 4); // var1 값을 획득합니다.
GETL(b, m + bp - 8); // var2 값을 획득합니다.

ENDP
```

이는 프로시저에서 임의의 위치에 변수를 선언하더라도, 해당 변수의 위치와 스택 메모리의 시작 지점의 값이 바뀌지 않음을 이용한 것이다. (bp-4)는 언제나 var1이며, (bp-8)은 언제나 var2이다. 이로써 같은 변수를 가리키기 위해 매번 sp를 추적해야 할 필요가 말끔히 사라졌다. 따라서 우리는 지역 변수에 접근할 때 sp 변수가 아닌 bp 변수를 이용해서 접근하는 것이 바람직하다.

### 3.8) 스택 포인터와 프로시저

방금 스택 포인터 변수 sp는 프로시저와도 관계가 있다고 했다. 정확히 어떤 관계일까? 이를 이해하기 위해 먼저 스택 메모리가 왜 스택 메모리인지부터 알아야겠다. 이를 위해 먼저 스택 메모리에 관한 명령을 보이겠다.

- PUSH(param): param 값을 스택 메모리에 푸시 한다.
- POP(param): 스택 메모리에서 팝 한 값을 param에 저장한다.

이전까지 우리는 PUSH를 함수를 호출하기 위해 값을 인자로 전달할 때 사용하는 명령으로 알고 있었다. 하지만 위에서 말했듯 실제로 PUSH는 스택 메모리에 값을 푸시 하는 역할을 하는데, 이것이 어떤 식으로 수행되는지를 알아보자. 다음은 이를 설명하기 위한 예제 코드다.

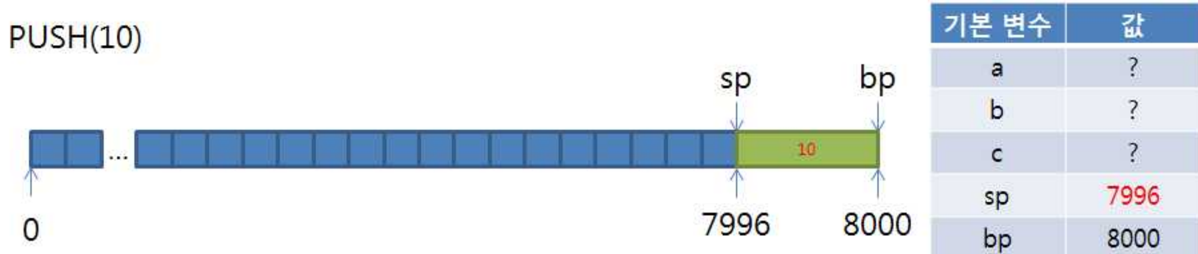
PushPop.c

```
#include "CIL.h"

// main
PROC(main)

PUSH(10)
PUSH(20)
PUSH(30)
POP(a)
POP(b)
POP(c)

ENDP
```

별도로 주석을 달지 않았지만, 스택을 이해하고 있는 여러분이라면 전혀 어렵다고 느끼지 않았을 것이다. 먼저 main 프로시저가 호출되면 다음 상태가 된다.

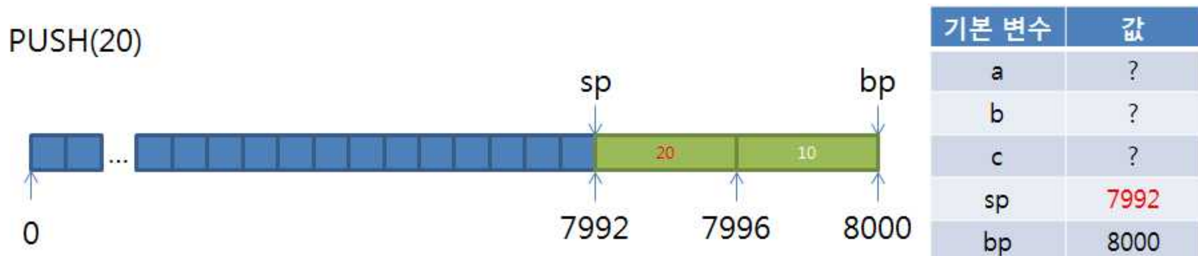
| 기본 변수 | 값 |
|---|---|
| a | ? |
| b | ? |
| c | ? |
| sp | 8000 |
| bp | 8000 |

10을 푸시 한다.




| 기본 변수 | 값 |
|---|---|
| a | ? |
| b | ? |
| c | ? |
| sp | 7996 |
| bp | 8000 |

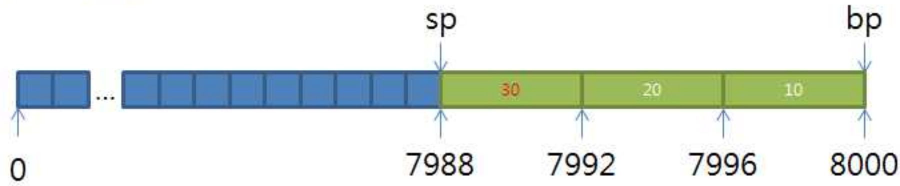
20을 푸시 한다.




| 기본 변수 | 값 |
|---|---|
| a | ? |
| b | ? |
| c | ? |
| sp | 7992 |
| bp | 8000 |

30을 푸시 한다.

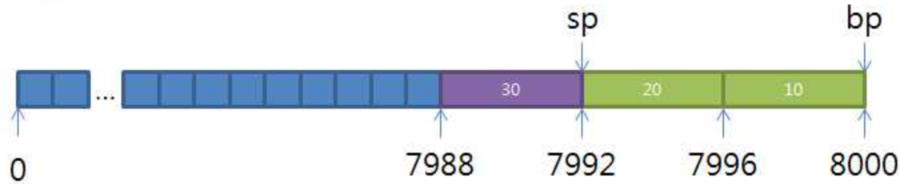
PUSH(30)




| 기본 변수 | 값 |
|---|---|
| a | ? |
| b | ? |
| c | ? |
| sp | 7988 |
| bp | 8000 |

팝 한 값을 a에 저장한다.

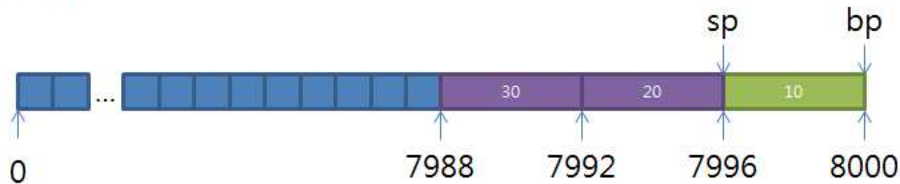
POP(a)




| 기본 변수 | 값 |
|---|---|
| a | 30 |
| b | ? |
| c | ? |
| sp | 7992 |
| bp | 8000 |

팝 한 값을 b에 저장한다.

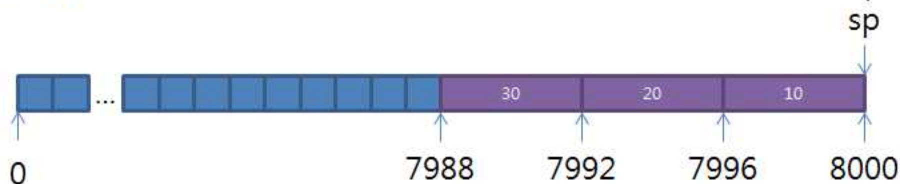
POP(b)




| 기본 변수 | 값 |
|---|---|
| a | 30 |
| b | 20 |
| c | ? |
| sp | 7996 |
| bp | 8000 |

팝 한 값을 c에 저장한다.

POP(c)




| 기본 변수 | 값 |
|---|---|
| a | 30 |
| b | 20 |
| c | 10 |
| sp | 8000 |
| bp | 8000 |

그림으로 보면 아주 간단한데, 혹시 스택 포인터의 움직임을 자세히 보았는가? 그렇다면 이것이 왼쪽 방향으로 향하는 배열 기반의 스택임을 알 수 있을 것이다. 우리가 앞으로 구현할 컴파일러는 메모리를 바이트 배열로 생각하고, 스택 영역은 위와 같이 배열 스택으로 간주한다. 위 과정에 익숙해져야 앞으로 프로젝트를 진행할 수 있다.

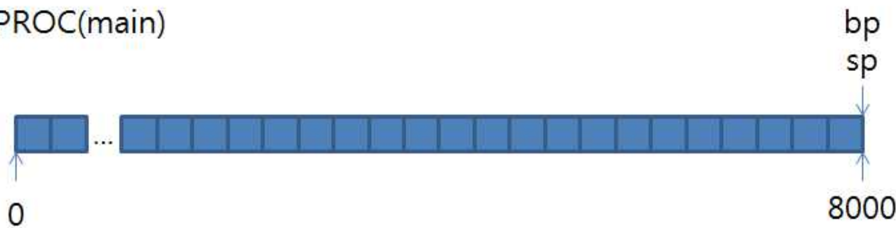
이제 지역 변수와 스택 메모리 사이의 관계를 이해했으니 다음을 보자. 다음은 10을 출력하는 단순한 프로그램의 코드다.

SpProc.c

```
#include "CIL.h"

// main
PROC(main)

// get_sum2(10, 20)을 호출합니다.
PUSH(20)
```

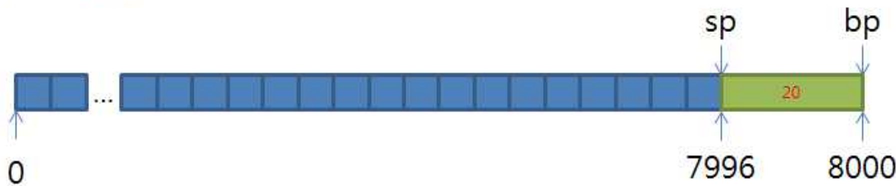
```
PUSH(10)
INVOKE(get_sum2)

// 반환된 값을 출력합니다.
PUSH(a)
INVOKE(print_int)

ENDP

// get_sum2
PROC(get_sum2)

// d에 두 번째 인자의 값을 대입합니다.
GETL(d, m+bp+12)
// a에 첫 번째 인자의 값을 대입합니다.
GETL(a, m+bp+8)
// a += d;
ADD(a, d)
// 함수 종료 시에는 a의 값이 항상 반환됩니다.
// return a;
RETURN()

ENDP
```

참고로 꽤 길다. 프로시저가 호출되면 다음 상태가 된다.

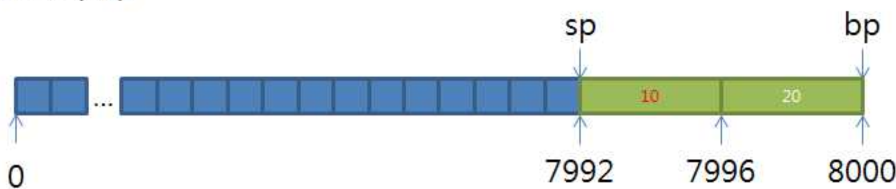



| 기본 변수 | 값 |
|---|---|
| a | ? |
| d | ? |
| sp | 8000 |
| bp | 8000 |

인자를 스택에 모두 푸시하고 프로시저를 호출한다.




| 기본 변수 | 값 |
|---|---|
| a | ? |
| d | ? |
| sp | 7996 |
| bp | 8000 |




| 기본 변수 | 값 |
|---|---|
| a | ? |
| d | ? |
| sp | 7992 |
| bp | 8000 |

INVOKE(get_sum2)

| 기본 변수 | 값 |
|---|---|
| a | ? |
| d | ? |
| sp | 7984 |
| bp | 7984 |

기본적으로 프로시저를 호출하면, 프로시저가 종료되었을 때 어느 위치부터 프로그램을 다시 실행해야 하는가, 즉 다음 명령의 주소 값을 넣어야 한다. 또한, 프로시저 시작 시에 새롭게 스택의 시작 지점이 정의되도록 bp 기본 변수의 값을 수정해야 한다. 일단 복귀 주소로 어떻게 돌아가는지에 대해서는 지금 설명하지 않겠다. 다음은 get_sum2 프로시저 내부에서 일어나는 루틴이다.

GETL(d, m+bp+12)

| 기본 변수 | 값 |
|---|---|
| a | ? |
| d | 20 |
| sp | 7984 |
| bp | 7984 |

GETL(d, m+bp+8)

| 기본 변수 | 값 |
|---|---|
| a | 10 |
| d | 20 |
| sp | 7984 |
| bp | 7984 |

ADD(a, d)

| 기본 변수 | 값 |
|---|---|
| a | 30 |
| d | 20 |
| sp | 7984 |
| bp | 7984 |

이 과정이 모두 끝나면 RETURN 명령을 이용해 get_sum2 프로시저를 호출한 main 프로시저로 복귀해야 한다.

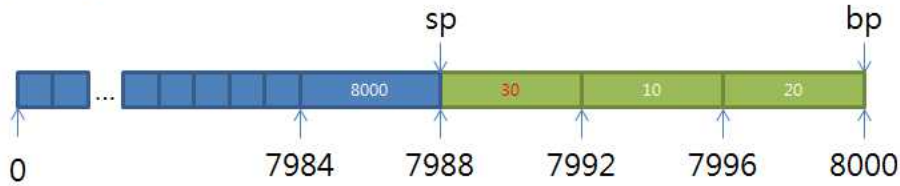
RETURN()

| 기본 변수 | 값 |
|---|---|
| a | 30 |
| d | 20 |
| sp | 7992 |
| bp | 8000 |

먼저 마지막으로 스택에 들어간 원소에서 팝 연산을 수행하여 bp를 스택의 복귀 주소로 맞춘다. 그 다음 다시 팝 연산을 수행하여 복귀 주소를 획득하고 해당 주소로 점프한다. 따라서 팝 연산을 2번 수행하므로 sp는 4바이트 * 2 = 8바이트만큼 이동하게 된다.

나머지는 반환 값을 출력하는 부분에 대한 것이며, 위의 그림을 이해하는 데 도움이 될 것이다.

PUSH(a)




| 기본 변수 | 값 |
|---|---|
| a | 30 |
| d | 20 |
| sp | 7988 |
| bp | 8000 |

INVOKE(print_int)

| 기본 변수 | 값 |
|---|---|
| a | 30 |
| d | 20 |
| sp | 7980 |
| bp | 7980 |

after print_int

| 기본 변수 | 값 |
|---|---|
| a | ? |
| d | 20 |
| sp | 7988 |
| bp | 8000 |

ENDP

| 기본 변수 | 값 |
|---|---|
| a | ? |
| d | 20 |
| sp | 8000 |
| bp | 8000 |

이와 같이 INVOKE 매크로와 RETURN 매크로로 프로시저를 호출하고 원래 주소로 복귀하는 방법을 알아보았다. 다만 완전히 설명이 끝난 것은 아닌데, 지금은 너무 많은 내용을 배웠으므로 단원을 잘라서 한 번 심호흡을 한 다음에 학습을 계속하자.

이제 심호흡이 끝났을 테니 다음으로 넘어갈 수 있겠다.

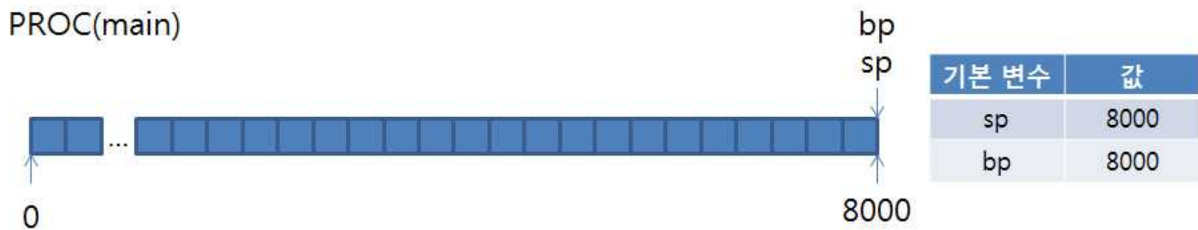
**3.9) 스택 포인터 정리하기**

다음은 10과 20을 출력하는 단순한 코드다.

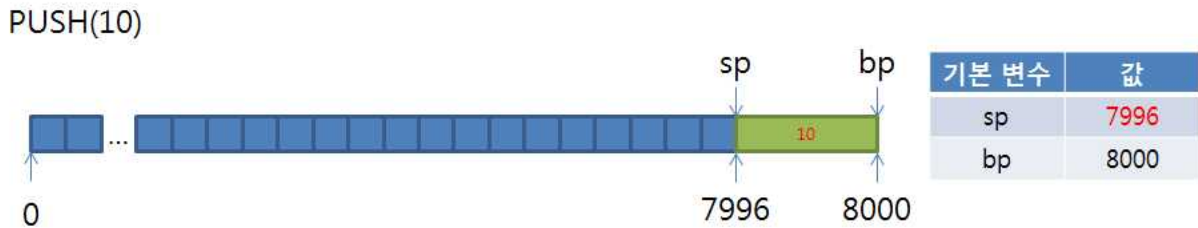
```
SpArrange.c
```

```
#include "CIL.h"

PROC(main)

PUSH(10)
INVOKE(print_int)

PUSH(20)
INVOKE(print_int)

ENDP
```

개행 명령이 없으므로 결과가 1020으로 나온다는 것만 빼면 설명할 것이 없는 단순한 코드다. 그런데 이 코드의 흐름을 한 번 살펴보자. 먼저 main 프로시저가 호출된 직후의 상황이다.



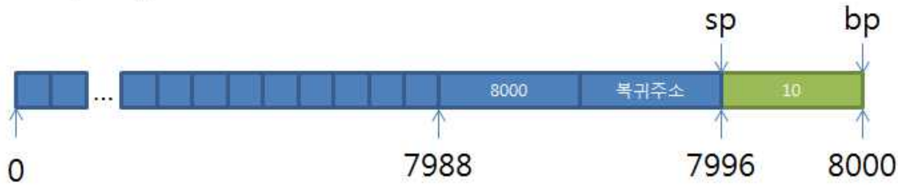

10을 푸시 한다.




print_int 프로시저를 호출하면 이렇게 될 것이고,

print_int 프로시저가 끝나면 이런 상태일 것이다.

after print_int




| 기본 변수 | 값 |
|---|---|
| sp | 7996 |
| bp | 8000 |

그런데 여기서 20을 푸시하면 이런 상태가 된다.

PUSH(20)

| 기본 변수 | 값 |
|---|---|
| sp | 7992 |
| bp | 8000 |

여기서 print_int 프로시저를 다시 호출하면 이렇게 되고,

INVOKE(print_int)

| 기본 변수 | 값 |
|---|---|
| sp | 7984 |
| bp | 7984 |

프로시저가 끝나면 이런 상태가 된다.

after print_int

| 기본 변수 | 값 |
|---|---|
| sp | 7992 |
| bp | 8000 |

결과적으로 함수 호출이 끝날 때마다 인자로 넘겼던 메모리가, 프로시저가 종료하기 전까지 계속 놀고 있는 상태가 된다. 고작 4바이트라고 생각할 수도 있으나, 만약 크기가 큰 구조체를 통째로 인자로 넘기는 경우에는 이는 분명한 문제가 된다. 따라서 이는 해결해야 하는 문제인데 어떻게 해야 할까?

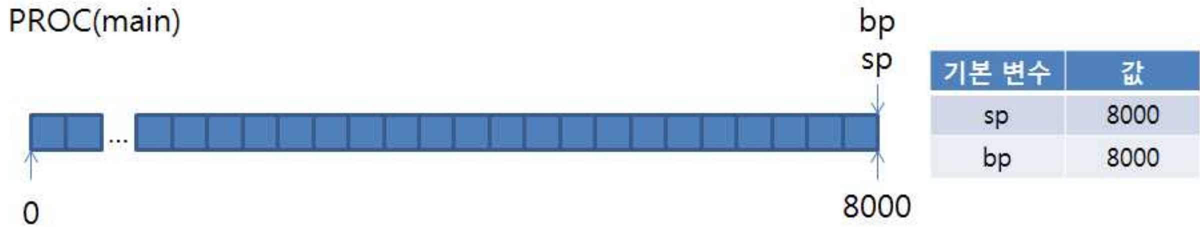
답은 매우 단순한데, 넘겼던 인자의 크기만큼 스택에서 메모리를 확보했으니, 넘겼던 인자의 크기만큼을 다시 스택 메모리에서 빼내면 된다. 즉 다음과 같이 해결하면 된다.

SpSolution.c

```
#include "CIL.h"

PROC(main)

PUSH(10)
INVOKE(print_int)
ADD(sp, 4) // 넘긴 인자의 크기만큼 스택 포인터에 더한다

PUSH(20)
```

```
INVOKE(print_int)
ADD(sp, 4) // 넘긴 인자의 크기만큼 스택 포인터에 더한다

ENDP
```

이 해법이 옳은지를 점검해보자. main 프로시저가 호출되면 다음 상태에 있게 된다.




| 기본 변수 | 값 |
|---|---|
| sp | 8000 |
| bp | 8000 |

print_int 함수가 종료될 때까지는 진행 상황이 같다. 그림으로는 다음 상태와 같다.

| 기본 변수 | 값 |
|---|---|
| sp | 7996 |
| bp | 8000 |

여기서 ADD 명령으로 sp를 조정하면 다음과 같은 일이 일어난다.

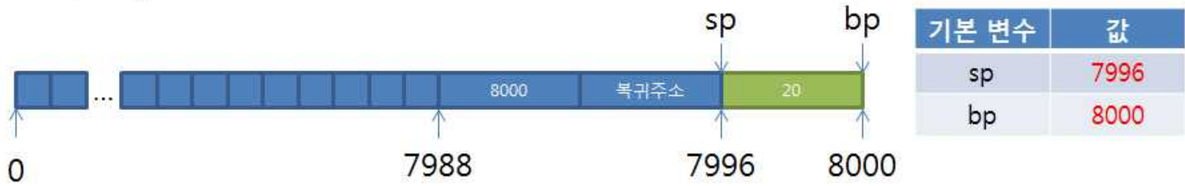
| 기본 변수 | 값 |
|---|---|
| sp | 8000 |
| bp | 8000 |

즉 main 프로시저를 처음으로 호출한 순간과 상황이 완전히 같아졌다. 이후에 진행되는 상황은 인자로 20이 넘어갔다는 사실을 제외하면 달라지지 않는다.

| 기본 변수 | 값 |
|---|---|
| sp | 7996 |
| bp | 8000 |

| 기본 변수 | 값 |
|---|---|
| sp | 7988 |
| bp | 7988 |

after print_int




ADD(sp, 4)

이와 같이 프로시저에 넘긴 인자의 크기만큼을 프로시저 종료 시에 스택 포인터에 더하는 해법이 적절함을 알 수 있었다.

**3.10) CALL 명령과 RET 명령을 이용한 프로시저 호출과 복귀**

지금까지는 프로시저를 호출할 때 INVOKE 매크로를, 프로시저를 호출한 곳으로 복귀할 때는 RETURN 매크로를 사용했다. 그렇게 강조해서 표현하지 않았지만, 사실 이 둘은 명령이 아니다. 지금까지 프로시저 호출에 사용한 INVOKE는 CALL 명령과 POP 명령의 조합으로 이루어져있는 매크로다. 복귀에 사용하는 RETURN은 프로시저 종료 시에 자동으로 메모리를 정리하도록 구현되어있다. 우리가 앞으로 구현할 컴파일러는 CALL 명령을 이용하는데, 그 이유는 INVOKE 매크로가 CALL 명령을 이용하여 구현되었기 때문이다. 여기서는 CALL과 RET 명령을 이용하여 프로시저를 다뤄보자. 이 내용은 중요하기 때문에 잘 이해하고 있어야 한다.

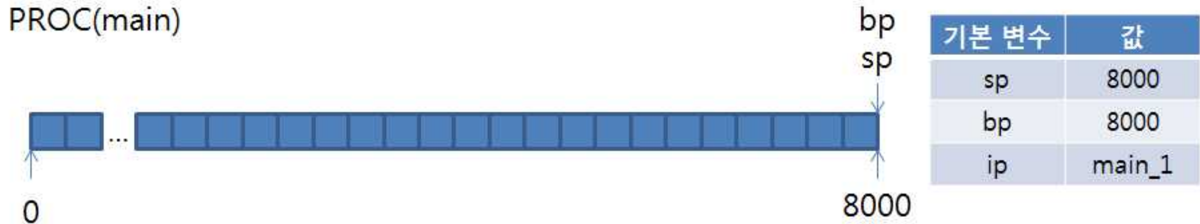
다음은 프로시저를 CALL 명령과 RET 명령을 이용하여 사용하는 코드다.

ProcNaked.c

```
#include "CIL.h"
STRING sHelloWorld = "Hello, world!\n";

PROC(main)

// naked_proc 프로시저를 호출합니다.
CALL(naked_proc)

ENDP

// NAKED 프로시저를 정의합니다.
PROC_NAKED(naked_proc)
PUSH(bp) // 이전 스택 시작 주소를 푸시하여 보관합니다.
MOVL(bp, sp) // 스택 시작 주소를 현재 스택 포인터로 맞춥니다.

PUSH(sHelloWorld)
INVOKE(print_str)
ADD(sp, 4)
```

```
MOVL(sp, bp) // 현재 스택 포인터를 스택 시작 주소로 맞춥니다.
POP(bp) // 보관했던 이전 스택 시작 주소를 불러옵니다.
RET() // 복귀 지점으로 돌아갑니다.
ENDP_NAKED
```

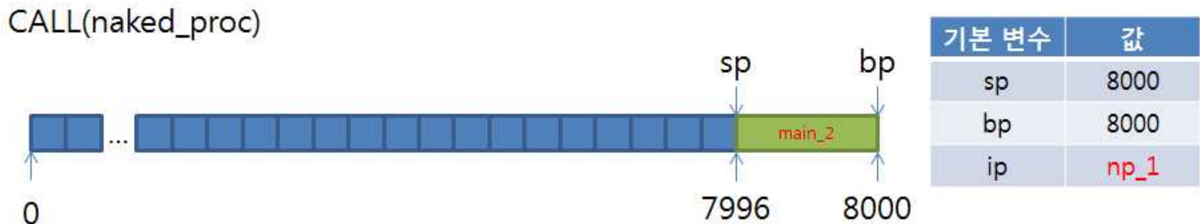
여기서는 PROC_NAKED 키워드를 사용하여 프로시저를 정의하고 있다. 이를 자세히 들여다보자. main 프로시저가 호출된 직후 메모리 상태는 다음과 같다.




| 기본 변수 | 값 |
|---|---|
| sp | 8000 |
| bp | 8000 |
| ip | main_1 |

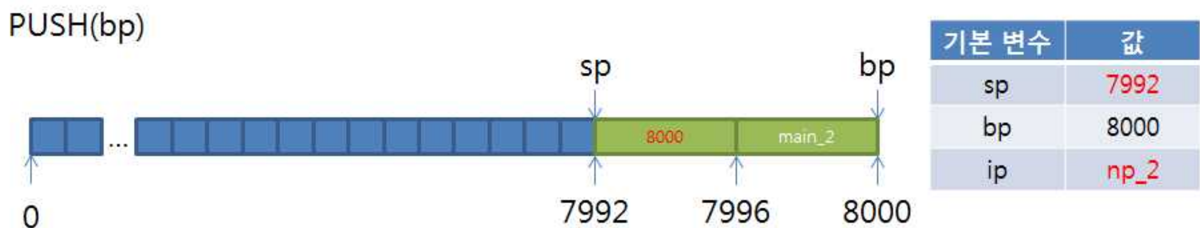
ip는 이후에 실행할 명령의 위치를 기록하는 숨겨진 기본 변수다. 값이 main_1인 건 main 프로시저에서 다음에 수행할 명령이 main_1이라는 뜻이다.

이제 CALL 명령을 이용해 naked_proc 프로시저를 호출하면, 현재 명령의 다음 위치가 스택에 푸시되고 naked_proc 프로시저로 진입한다.



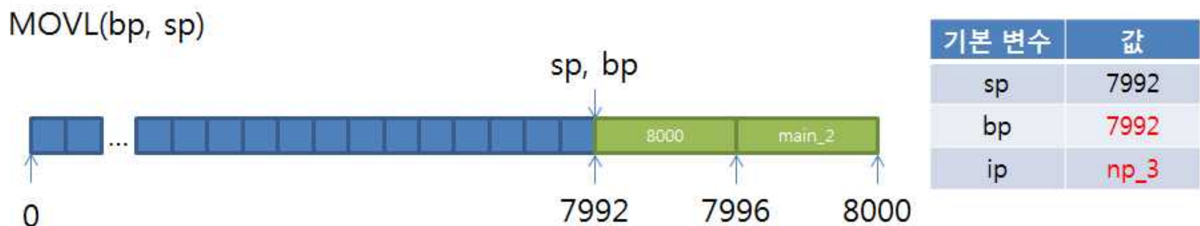

| 기본 변수 | 값 |
|---|---|
| sp | 8000 |
| bp | 8000 |
| ip | np_1 |

main_2가 푸시 된 건, main_1은 CALL 명령이었으니 그 다음에 수행할 명령을 푸시해야 하기 때문이다. 이후 bp를 스택에 푸시 하는데, 주석에도 쓰여 있지만 이는 이전 프로시저의 스택 시작 지점을 보관하기 위함이다.




| 기본 변수 | 값 |
|---|---|
| sp | 7992 |
| bp | 8000 |
| ip | np_2 |

이제 bp에 현재 스택 포인터 sp의 값을 대입하여 스택 시작 주소를 갱신한다.



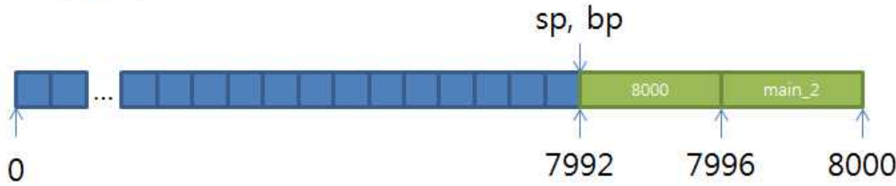

| 기본 변수 | 값 |
|---|---|
| sp | 7992 |
| bp | 7992 |
| ip | np_3 |

바로 여기까지가 INVOKE 매크로가 수행하는 과정이다. INVOKE 매크로는 내부적으로 스택을 알아서 정리하는 기능을 가지고 있기 때문에 실제로 사용하기엔 더 편하지만, 만약 당신이 프로그램을 디버깅할 일이 생겨서 디스어셈블리 파일을 분석하게 된다면 모든 프로시저의 시작은 이 형태로 나타난다. 매크로의 내부가 그렇게 어렵지 않고, 앞으로 진행할 때도 INVOKE가 아닌 CALL을 기준으로 구현할 것이

다. 따라서 반드시 이 형태에 익숙해져야 한다.

이제 프로시저의 호출을 보았으니, 이전 프로시저로 복귀하는 모습을 볼 차례다. print_str 프로시저 호출에 대해서는 이제 모두 알고 있으리라 생각하므로, ADD 명령이 수행되었을 때의 메모리 상태만 보이겠다.

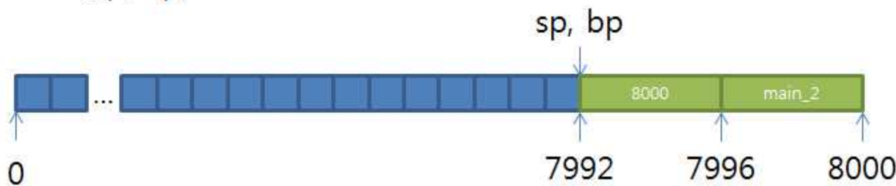
ADD(sp, 4)




| 기본 변수 | 값 |
|---|---|
| sp | 7992 |
| bp | 7992 |
| ip | np_6 |

사실 위에 보인 그림과 다르지 않다. 다음은 이전 프로시저에 복귀하기 위해 sp를 현재 스택 메모리의 시작으로 옮기는 과정이다.

MOVL(sp, bp)

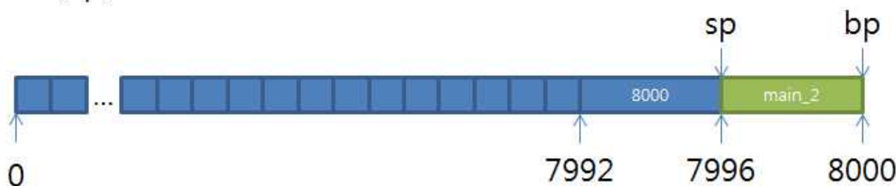



| 기본 변수 | 값 |
|---|---|
| sp | 7992 |
| bp | 7992 |
| ip | np_7 |

바뀐 것이 없는데, 왜냐하면 이 프로시저에서는 지역 변수가 선언되지 않았기 때문이다. sp를 bp로 맞춤으로써 해당 지역 변수를 모두 사용하지 않는 상태로 만드는 것인데, 지역 변수가 없으므로 상태가 변하지 않는다. 이는 다른 프로시저를 직접 그려가면서 학습하면 납득할 수 있다.

이제 POP 연산을 통해 bp에 이전 스택 시작 주소를 복원한다.

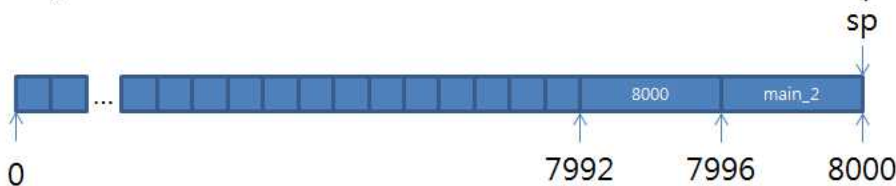
POP(bp)




| 기본 변수 | 값 |
|---|---|
| sp | 7996 |
| bp | 8000 |
| ip | np_8 |

POP 연산이 sp도 움직이고 있음에 주목해야 한다. 이제 마지막으로 RET 명령을 이용하여 이전 프로시저로 복귀한다.

RET()




| 기본 변수 | 값 |
|---|---|
| sp | 8000 |
| bp | 8000 |
| ip | main_2 |

이로써 프로시저의 호출과 복귀가 모두 끝난다.

### 4. 그래서 어셈블리는 언제 가르쳐줄 겁니까?

이미 어셈블리를 공부한 사람은 느꼈겠지만, 사실 CIL은 C와 어셈블리의 문법적 중간 단계 언어로 볼 수 있다. 여기서 익힌 문법을 약간의 노력만 들여서 고치면 그것이 바로 어셈블리가 된다. 지금까지 어셈블리에 익숙해지기 위해 어셈블리와 비슷한 언어를 학습했으니, 이제 CIL과 실제 어셈블리의 문법적인 차이만을 위주로 어셈블리를 배울 수 있다.

하지만 이번에 많은 내용을 배웠다. 어셈블리에 이미 익숙한 사람이라면 위 내용을 이해하기 어렵지

않았을 것이다. 필자도 처음에는 한 문서에 어셈블리에 대한 내용을 한꺼번에 정리하고픈 욕심이 있었다. 그러나 이 문서를 읽기 위한 최소 조건은 C와 C++를 알고 있다는 것이고, 어셈블리를 처음 접한 사람이라면 기존의 프로그래밍 방식과는 사뭇 다른 방식에 이해하는 데 무리가 있지 않을까 생각이 들어서, 일단 이 정도로 어셈블리를 배우기 위한 기본적인 내용만 이해하고 다음 문서에서 문법적인 차이를 설명하는 것이 좋겠다는 생각이 들었다.

결론을 말하자면, 본격적인 어셈블리에 대한 설명은 다음 문서로 미룬다. 혹 어셈블리에 대해 먼저 공부하고 싶다면 다음 링크의 문서를 내려받아 읽으면 도움이 될 것이다. 원문이 영어인데, 스크롤을 가장 아래로 내리면 한국의 이재범 님께서 이 문서를 한국어로 번역해놓으셨으니 이 문서를 다운로드하여 한국어 문서를 읽을 수 있다.

초보자를 위한 PC 어셈블리어: http://www.drpaulcarter.com/pcasm/

5. 단원 마무리

어셈블리를 가르친다면서 한참동안 CIL이라는 괴상한 언어를 가르쳤다. 중간 언어가 필요하다는 것은 이해했지만 굳이 별로 믿기지도 않는 무명 프로그래머가 만든 언어로 강의를 진행해야했을까 싶었던 분들을 위해 마무리에서 이야기하자면, 어셈블리를 바로 가르치지 않고 굳이 CIL이라는 중간 단계 언어를 도입한 것은, 어셈블리 언어를 당장 배우기에는 어셈블리에는 지금까지 설명한 문법적 요소 외에 불필요하게 알아야 할 지식들이 너무 많다는 이유가 있었다. 예를 들어 바로 어셈블리 코드를 시작하려면 컴파일러는 어떤 것을 사용해야 하는지, globl은 무슨 키워드인지, .text는 뭐고 .data와 무슨 차이가 있는지, 심지어는 진입점이 어디인지조차도 제대로 갈피가 안 잡히는 경우가 많다. CIL은 C를 기반으로 한 언어이기 때문에 이런 면에서는 월등히 유리했다. 파일을 포함하려면 #include 전처리기, 전역 문자열을 선언하려면 STRING을 사용하면 된다. 즉 CIL은 당장 필요하지 않은 부분은 알고 있는 대로, 정말 배워야 하는 부분은 모두 CIL로 작성하도록 하여 문서의 내용에 보다 집중할 수 있도록 필자가 만든 도구이다.

다음에 배울 내용은 실제 어셈블리 언어에 대한 내용이다. 처음에는 MASM이라는 Microsoft에 종속적인 어셈블리 언어를 기준으로 설명하려고 했는데, 기왕 문서를 분리하는 김에 이에 대해서도 검토해볼 기회를 갖게 되어 다행이라고 생각한다.