☆어셈블리★

2019/01/14
Team SCP
정재훈

목차

1. 어셈블리어와 명령어의 구조
2. CPU와 메모리
3. 레지스터 (고리타분하게 설명할거 아니니까 겁먹 ㄴㄴ)
4. pwnable.kr leg

Assembly Language

- 짧게 어셈
- 기계가 인식할 수 있는 기계어임

어셈블리어의 구조

OPCODE	OPERAND....
연산자 (명령어)	피연산자 (여러 개)

x86 Opcode Structure and Instruction Overview

1st \ 2nd	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD						ES PUSH	ES POP	OR						CS PUSH	TWO BYTE
1	ADC						PUSH SS	POP SS	SBB						DS	POP DS
2	AND						ES SEGMENT OVERRIDE	DAA	SUB						CS SEGMENT OVERRIDE	DAS
3	XOR						SS	AAA	CMP						DS	AAS
4	INC								DEC							
5	PUSH								POP							
6	PUSHAD	POPAD	BOUND	ARPL	FS (SEGMENT OVERRIDE)	GS (SEGMENT OVERRIDE)	OPERAND SIZE (SIZE OVERRIDE)	ADDRESS SIZE (SIZE OVERRIDE)	PUSH	IMUL	PUSH	IMUL	INS		OUTS	
7	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
	Jcc															
8	ADD/ADC/AND/XOR OR/SBB/SUB/CMP				TEST		XCHG		MOV REG				MOV SREG	LEA	MOV SREG	POP
9	NOP / XCHG EAX								CWD	CDQ	CALLF	WAIT	PUSHFD	POPFD	SAHF	LAHF
A	MOV EAX				MOVS		CMPS		TEST		STOS		LODS		SCAS	
B	MOV															
C	SHIFT IMM		RETN		LES	LDS	MOV IMM		ENTER	LEAVE	RETF		INT3	INT IMM	INTO	IRETD
D	SHIFT 1		SHIFT CL		AAM	AAD	SALC	XLAT	FPU							
	ROL/ROR/RCL/RCR/SHL/SHR/SAL/SAR															
E	LOOPNZ	LOOPZ	LOOP	JECXZ	IN IMM		OUT IMM		CALL	JMP	JMPF	JMP SHORT	IN DX		OUT DX	
	CONDITIONAL LOOP															
F	LOCK EXCLUSIVE ACCESS	ICE BP	REPNE	REPE	HLT	CMC	TEST/NOT/NEG [i]MUL/[i]DIV		CLC	STC	CLI	STI	CLD	STD	INC DEC	INC/DEC CALL/JMP PUSH
			CONDITIONAL REPETITION													

1st \ 2nd	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	{L,S}LDT {L,S}TR VER{R,W}	{L,S}GDT {L,S}IDT {L,S}MSW	LAR	LSL			CLTS		INVD	WBINVD		UD2		NOP		
1	*SSE{1,2,3}*								Prefetch *SSE1*	HINT_NOP						
2	MOV CR/DR								*SSE{1,2}*							
3	WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		GETSEC *SMX*	MOVBE / THREE BYTE		THREE BYTE *SSE4*					
4	CMOV															
5	*SSE{1,2}*															
6	*MMX, SSE2*															
7	*MMX, SSE{1,2,3}, VMX*												*MMX, SSE{2,3}*			
8	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
	Jcc SHORT															
9	SETO	SETNO	SETB	SETNB	SETE	SETNE	SETBE	SETA	SETS	SETNS	SETPE	SETPO	SETL	SETGE	SETLE	SETG
	SETcc															
A	PUSH FS	POP FS	CPUID	BT	SHLD				PUSH GS	POP GS	RSM	BTS	SHRD		*FENCE	IMUL
B	CMPXCHG		LSS	BTR	LFS	LGS	MOVZX		POPCNT	UD	BT BTS BTR BTC	BTC	BSF	BSR	MOVSX	
C	XADD		*SSE{1,2}*					CMPXCHG	BSWAP							
D	*MMX, SSE{1,2,3}*															
E	*MMX, SSE{1,2}*															
F	*MMX, SSE{1,2,3}*															

Arithmetic & Logic

Memory

Stack

Control Flow & Conditional

Prefix

System & I/O

No Operation (NOP) / Multiple Instructions / Extended Instruction Set

General Opcode Structure

Element Information # of bytes	Prefix 0-4	Opcode 1-3	AddrMode (mod, reg, r/m) 0-1	SIB Byte (scale, index, base) 0-1	Displacement 0/1/2/4	Immediate Data 0/1/2/4
Bit structure		O O O O O O D L	MOD MOD REG REG REG RM RM RM	S S I I I B B B		

Main Opcode bits
Direction bit
Operand length bit
r/m field
Register/Opcode modifier, defined by primary opcode
Addressing mode
Base field
Index field
Scale field

Addressing Modes

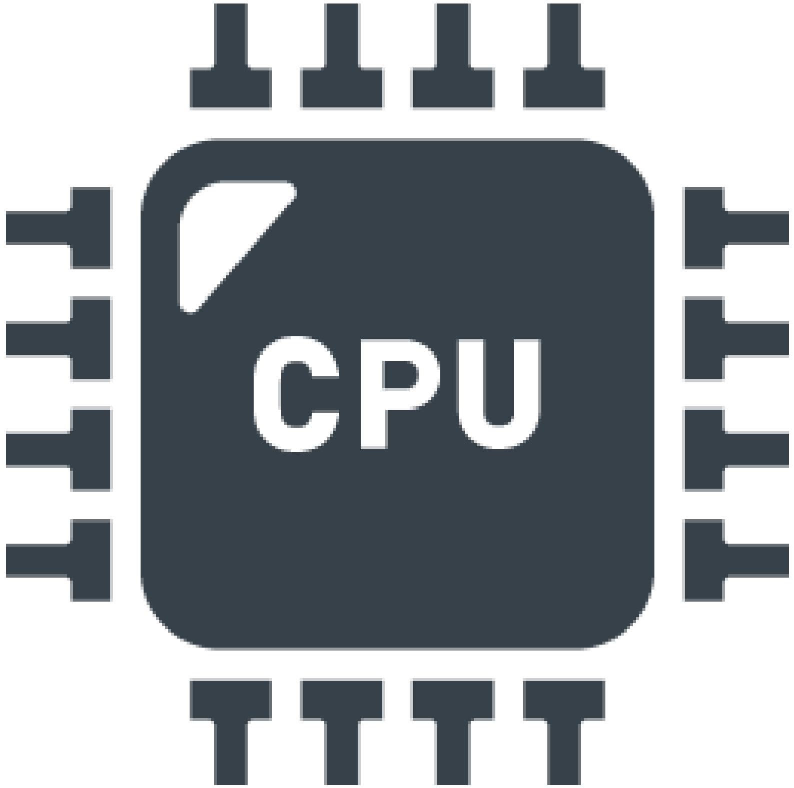
mod	00		01		10		11
r/m	16bit	32bit	16bit	32bit	16bit	32bit	r/m // REG
000	[BX+SI]	[EAX]	[BX+SI]+disp8	[EAX]+disp8	[BX+SI]+disp16	[EAX]+disp32	AL / AX / EAX
001	[BX+DI]	[ECX]	[BX+DI]+disp8	[ECX]+disp8	[BX+DI]+disp16	[ECX]+disp32	CL / CX / ECX
010	[BP+SI]	[EDX]	[BP+SI]+disp8	[EDX]+disp8	[BP+SI]+disp16	[EDX]+disp32	DL / DX / EDX
011	[BP+DI]	[EBX]	[BP+DI]+disp8	[EBX]+disp8	[BP+DI]+disp16	[EBX]+disp32	BL / BX / EBX
100	[SI]	*SIB*	[SI]+disp8	*SIB*+disp8	[SI]+disp16	*SIB*+disp32	AH / SP / ESP
101	[DI]	disp32	[DI]+disp8	[EBP]+disp8	[DI]+disp16	[EBP]+disp32	CH / BP / EBP
110	disp16	[ESI]	[BP]+disp8	[ESI]+disp8	[BP]+disp16	[ESI]+disp32	DH / SI / ESI
111	[BX]	[EDI]	[BX]+disp8	[EDI]+disp8	[BX]+disp16	[EDI]+disp32	BH / DI / EDI

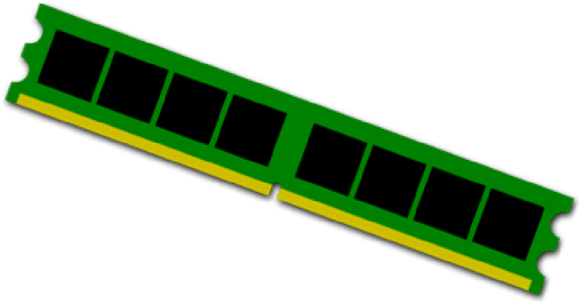
SIB Byte Structure

encoding	scale (2bit)	Index (3bit)	Base (3bit)
000	$2^0=1$	[EAX]	EAX
001	$2^1=2$	[ECX]	ECX
010	$2^2=4$	[EDX]	EDX
011	$2^3=8$	[EBX]	EBX
100	--	none	ESP
101	--	[EBP]	disp32 / disp8+ [EBP] / disp32 + [EBP]
110	--	[ESI]	ESI
111	--	[EDI]	EDI
SIB value = index * scale + base			

v1.0 – 30.08.2011
Contact: Daniel Plohmann – +49 228 73 54 228 – daniel.plohmann@fkie.fraunhofer.de

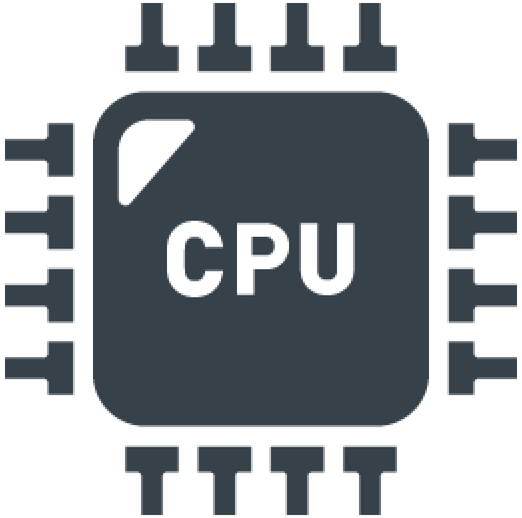
Source: Intel x86 Instruction Set Reference
Opcode table presentation inspired by work of Ange Albertini





CPU와 메모리

CPU

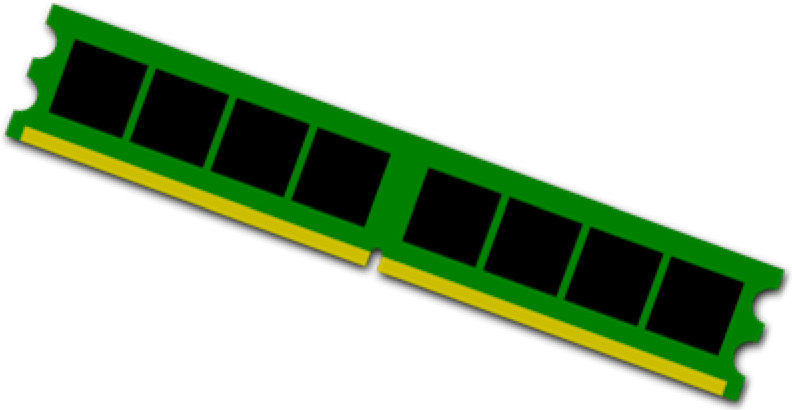


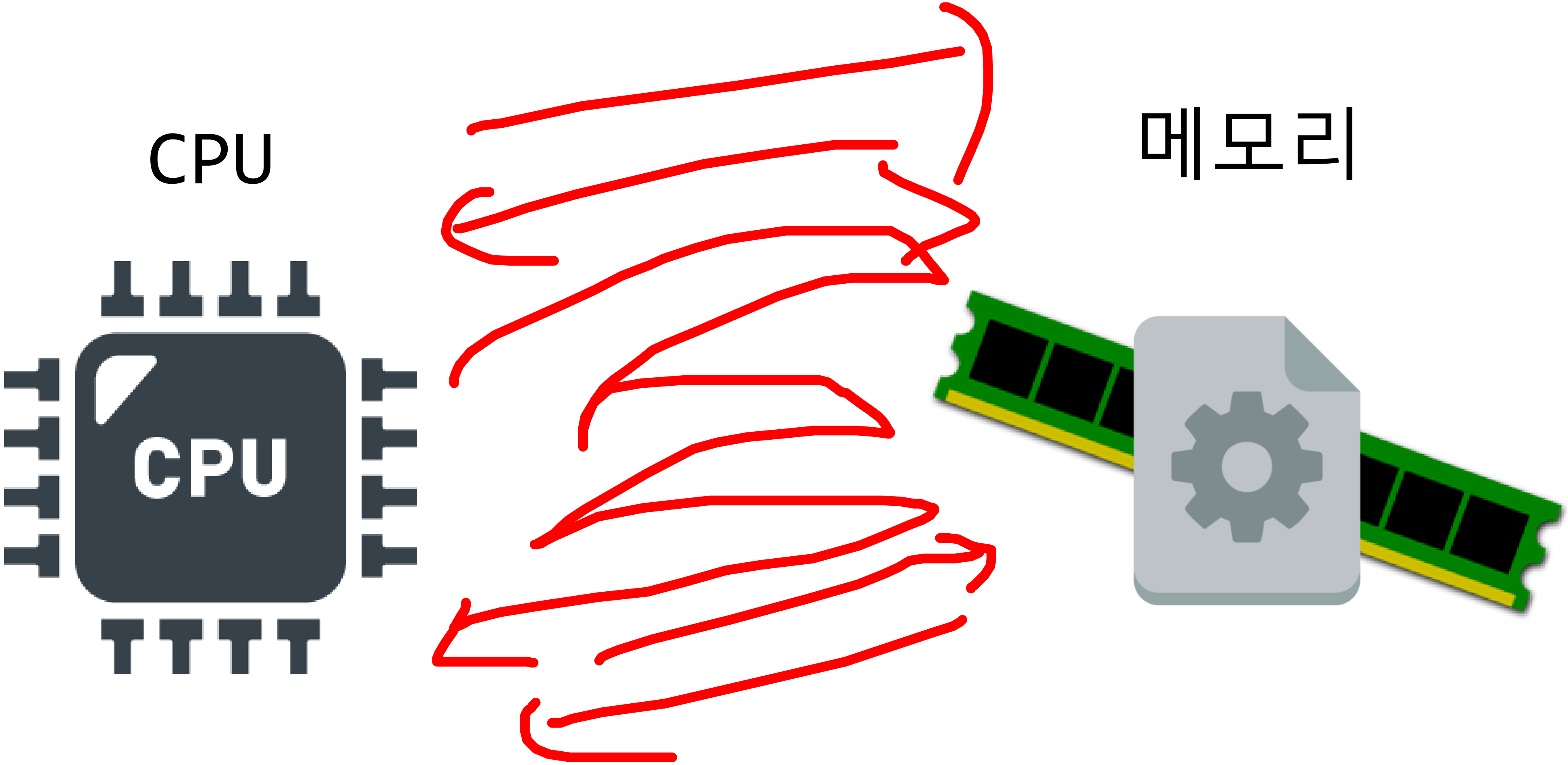
프로세서!

- 레지스터
- 연산장치
- 제어장치

메모리

- CPU 에서 즉각적으로 수행할 프로그램과 데이터를 저장하거나 프로세서에서 처리한 결과를 메인메모리에 저장한다

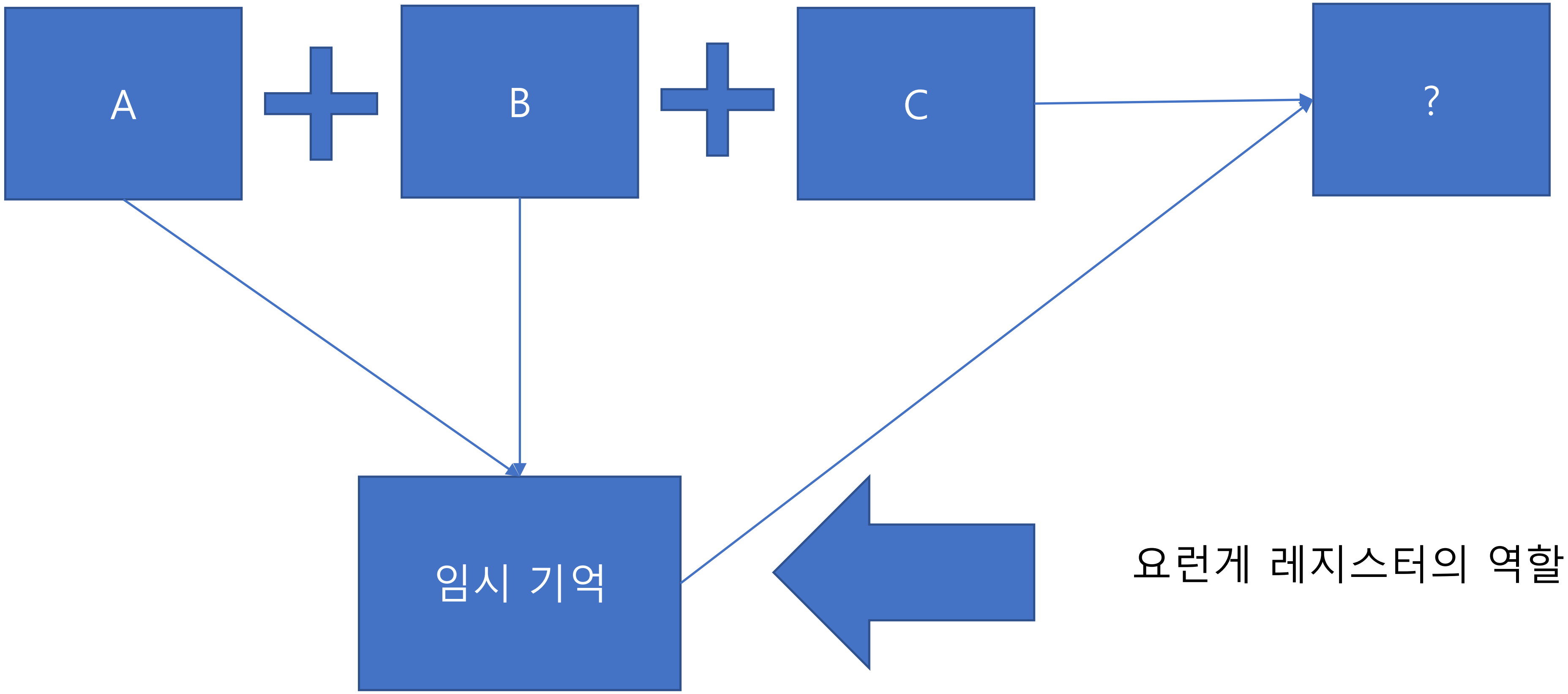



CPU
메모리
CPU

메모리에 올라가있는 프로그램

```
(gdb) disass key1
Dump of assembler code for function key1:
   0x00008cd4 <+0>:     push    {r11}           ; (str r11, [sp, #-4]!)
   0x00008cd8 <+4>:     add     r11, sp, #0
   0x00008cdc <+8>:     mov     r3, pc
   0x00008ce0 <+12>:    mov     r0, r3
   0x00008ce4 <+16>:    sub     sp, r11, #0
   0x00008ce8 <+20>:    pop     {r11}           ; (ldr r11, [sp], #4)
   0x00008cec <+24>:    bx      lr
End of assembler dump.
```

레지스터



요런게 레지스터의 역할

중요한,특별한 레지스터

ip!

- PC(Program Counter)
  - 다음에 실행할 명령어의 주소를 보관하는 레지스터

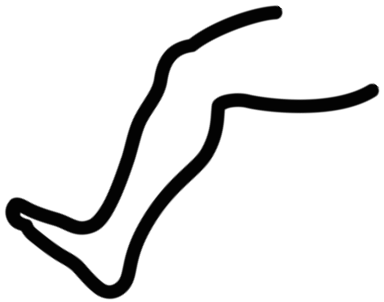
- LR(Link Register)
  - 함수 호출 전에 다시 되돌아가 실행할 주소를 보관

test

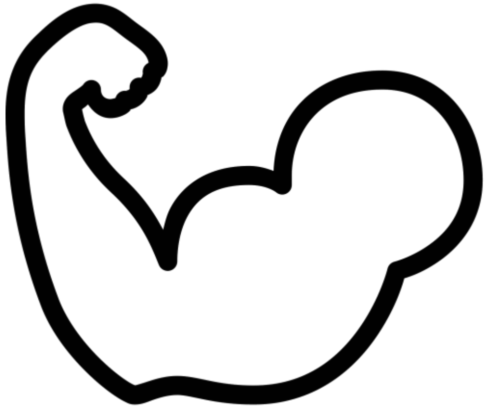
```
main:
0x10 adds  r0, #1
0x11 adds  r1, #2
0x12 add   r2, r1, r0
0x13 bl    0x08 <Func1>
0x14 mov   r4, r3
0x15 add   r4, r4, r2
```

```
Func1:
0x08 add   r3, #4
```

pwnable.kr leg



ARM Architecture



pwnable.kr leg

return 값

```
0x00008d68 <+44>:   bl    0x8cd4 <key1>
0x00008d6c <+48>:   mov   r4, r0
0x00008d70 <+52>:   bl    0x8cf0 <key2>
0x00008d74 <+56>:   mov   r3, r0
0x00008d78 <+60>:   add   r4, r4, r3                        r4 = KEY1( ) + KEY2( )
0x00008d7c <+64>:   bl    0x8d20 <key3>
0x00008d80 <+68>:   mov   r3, r0
0x00008d84 <+72>:   add   r2, r4, r3                        r2 = r4 + KEY3( )
0x00008d88 <+76>:   ldr   r3, [r11, #-16]
0x00008d8c <+80>:   cmp   r2, r3
0x00008d90 <+84>:   bne   0x8da8 <main+108>
0x00008d94 <+88>:   ldr   r0, [pc, #44]    ; 0x8dc8 <main+140>
0x00008d98 <+92>:   bl    0x1050c <puts>
0x00008d9c <+96>:   ldr   r0, [pc, #40]    ; 0x8dcc <main+144>
0x00008da0 <+100>:  bl    0xf89c <system>
0x00008da4 <+104>:  b     0x8db0 <main+116>
0x00008da8 <+108>:  ldr   r0, [pc, #32]    ; 0x8dd0 <main+148>
0x00008dac <+112>:  bl    0x1050c <puts>
```

pwnable.kr leg

KEY1의 r0 =

KEY2의 r0 =

KEY3의 r0 =

```
Dump of assembler code for function key1:
   0x00008cd4 <+0>:     push    {r11}           ; (str r11, [sp, #-4]!)
   0x00008cd8 <+4>:     add     r11, sp, #0
   0x00008cdc <+8>:     mov     r3, pc
   0x00008ce0 <+12>:    mov     r0, r3
   0x00008ce4 <+16>:    sub     sp, r11, #0
   0x00008ce8 <+20>:    pop     {r11}           ; (ldr r11, [sp], #4)
   0x00008cec <+24>:    bx      lr
End of assembler dump.
```

ARM에서 PC

1. 실행중인 명령어 (execute 단계)
2. 다음 명령어 (decode 단계)
3. 다다음 명령어 (fetch 단계)

arm에서 program counter는 fetch 단계의 명령어를 저장

pwnable.kr leg

KEY1의 r0 = 0x8ce4

KEY2의 r0 =

KEY3의 r0 =

```
Dump of assembler code for function key1:
   0x00008cd4 <+0>:     push    {r11}           ; (str r11, [sp, #-4]!)
   0x00008cd8 <+4>:     add     r11, sp, #0
   0x00008cdc <+8>:     mov     r3, pc
   0x00008ce0 <+12>:    mov     r0, r3
   0x00008ce4 <+16>:    sub     sp, r11, #0
   0x00008ce8 <+20>:    pop     {r11}           ; (ldr r11, [sp], #4)
   0x00008cec <+24>:    bx      lr
End of assembler dump.
```

pwnable.kr leg

KEY1의 r0 = 0x8ce4

KEY2의 r0 = 0x8d0c

KEY3의 r0 =

```
Dump of assembler code for function key2:
   0x00008cf0 <+0>:     push   {r11}           ; (str r11, [sp, #-4]!)
   0x00008cf4 <+4>:     add    r11, sp, #0
   0x00008cf8 <+8>:     push   {r6}            ; (str r6, [sp, #-4]!)
   0x00008cfc <+12>:    add    r6, pc, #1
   0x00008d00 <+16>:    bx     r6
   0x00008d04 <+20>:    mov    r3, pc
   0x00008d06 <+22>:    adds   r3, #4
   0x00008d08 <+24>:    push   {r3}
   0x00008d0a <+26>:    pop    {pc}
   0x00008d0c <+28>:    pop    {r6}            ; (ldr r6, [sp], #4)
   0x00008d10 <+32>:    mov    r0, r3
   0x00008d14 <+36>:    sub    sp, r11, #0
   0x00008d18 <+40>:    pop    {r11}           ; (ldr r11, [sp], #4)
   0x00008d1c <+44>:    bx     lr
End of assembler dump.
```

pwnable.kr leg

KEY1의 r0 = 0x8ce4

KEY2의 r0 = 0x8d0c

KEY3의 r0 = 0x8d28

```
Dump of assembler code for function key3:
   0x00008d20 <+0>:     push    {r11}           ; (str r11, [sp, #-4]!)
   0x00008d24 <+4>:     add     r11, sp, #0
   0x00008d28 <+8>:     mov     r3, lr
   0x00008d2c <+12>:    mov     r0, r3
   0x00008d30 <+16>:    sub     sp, r11, #0
   0x00008d34 <+20>:    pop     {r11}           ; (ldr r11, [sp], #4)
   0x00008d38 <+24>:    bx      lr
End of assembler dump.
```

pwnable.kr leg

return 값

```
0x00008d68 <+44>:    bl     0x8cd4 <key1>
0x00008d6c <+48>:    mov    r4, r0
0x00008d70 <+52>:    bl     0x8cf0 <key2>
0x00008d74 <+56>:    mov    r3, r0
0x00008d78 <+60>:    add    r4, r4, r3                          r4 = KEY1( ) + KEY2( )
0x00008d7c <+64>:    bl     0x8d20 <key3>
0x00008d80 <+68>:    mov    r3, r0
0x00008d84 <+72>:    add    r2, r4, r3                          r2 = r4 + KEY3( )
0x00008d88 <+76>:    ldr    r3, [r11, #-16]
0x00008d8c <+80>:    cmp    r2, r3
0x00008d90 <+84>:    bne    0x8da8 <main+108>
0x00008d94 <+88>:    ldr    r0, [pc, #44]      ; 0x8dc8 <main+140>
0x00008d98 <+92>:    bl     0x1050c <puts>
0x00008d9c <+96>:    ldr    r0, [pc, #40]      ; 0x8dcc <main+144>
0x00008da0 <+100>:   bl     0xf89c <system>
0x00008da4 <+104>:   b      0x8db0 <main+116>
0x00008da8 <+108>:   ldr    r0, [pc, #32]      ; 0x8dd0 <main+148>
0x00008dac <+112>:   bl     0x1050c <puts>
```

pwnable.kr leg

KEY1의 r0 = 0x8ce4

KEY2의 r0 = 0x8d0c

KEY3의 r0 = 0x8d80

+

0x1a770

pwnable.kr leg