



EDDI

Electronic Design
Development Institute

에디로봇아카데미 임베디드 마스터 Lv1 과정

제 5기

2023. 06. 08

박 상호

ARM Cross Compile

1. 컴파일

- 32-Bit ARM : `arm-linux-gnueabi-gcc -g -o (File Name) (File Name).c`
- 64-Bit ARM : `aarch64-linux-gnu-gcc -g -o (File Name) (File Name).c`

2. QEMU Emulator 실행

- 32-Bit ARM : `qemu-arm-static -g (Port Number) -L /usr/arm-linux-gnueabi (File Name)`
- 64-Bit ARM : `qemu-aarch64-static -g (Port Number) -L /usr/aarch64-linux-gnu (File Name)`

3. gdb-multiarch 실행 (별도 Terminal 실행)

- gdb-multiarch : Multi-Architecture gdb 실행 후 아래 명령어 입력
- 32-Bit ARM : `set arc arm`
- 64-Bit ARM : `set arc aarch64`
- `file (File Name)` : debug 할 파일 지정
- `target remote localhost:(Port Number)` : QEMU 연결
- `disas main` : main 함수 disassembly Entry Point 확인
- `b *(Entry Point Address)` : break point 생성
- `c` : continue

4. gdb-multiarch 종료

q 입력 or Ctrl + d

Stack Frame (ARM 32-Bit)

```
Dump of assembler code for function main:
=> 0x00010428 <+0>:  push    {r11, lr}
    0x0001042c <+4>:  add     r11, sp, #4
    0x00010430 <+8>:  sub     sp, sp, #8
    0x00010434 <+12>: mov     r3, #3
    0x00010438 <+16>: str     r3, [r11, #-12]
    0x0001043c <+20>: ldr     r0, [r11, #-12]
    0x00010440 <+24>: bl      0x10400 <multiply_two>
    0x00010444 <+28>: str     r0, [r11, #-8]
    0x00010448 <+32>: ldr     r1, [r11, #-8]
    0x0001044c <+36>: ldr     r0, [pc, #16] ; 0x10464 <main+60>
    0x00010450 <+40>: bl      0x102e0 <printf@plt>
    0x00010454 <+44>: mov     r3, #0
    0x00010458 <+48>: mov     r0, r3
    0x0001045c <+52>: sub     sp, r11, #4
    0x00010460 <+56>: pop     {r11, pc}
    0x00010464 <+60>: ldrdeq  r0, [r1], -r8
End of assembler dump.
```

```
Dump of assembler code for function main:
    0x00010428 <+0>:  push    {r11, lr}
    0x0001042c <+4>:  add     r11, sp, #4
    0x00010430 <+8>:  sub     sp, sp, #8
=> 0x00010434 <+12>: mov     r3, #3
    0x00010438 <+16>: str     r3, [r11, #-12]
    0x0001043c <+20>: ldr     r0, [r11, #-12]
    0x00010440 <+24>: bl      0x10400 <multiply_two>
    0x00010444 <+28>: str     r0, [r11, #-8]
    0x00010448 <+32>: ldr     r1, [r11, #-8]
    0x0001044c <+36>: ldr     r0, [pc, #16] ; 0x10464 <main+60>
    0x00010450 <+40>: bl      0x102e0 <printf@plt>
    0x00010454 <+44>: mov     r3, #0
    0x00010458 <+48>: mov     r0, r3
    0x0001045c <+52>: sub     sp, r11, #4
    0x00010460 <+56>: pop     {r11, pc}
    0x00010464 <+60>: ldrdeq  r0, [r1], -r8
End of assembler dump.
```

main 함수 호출

lr	r11 + 0x04 == r14 (Return Address)
r11	== r11 (Frame Pointer)
	r11 - 0x04
	r11 - 0x08
0x03	r11 - 0x0C == r13 (Stack Pointer) (str r3, [r11, #-12])

multiply_two 함수 호출 전 Register를 통한 num 인자 전달

r0	0x00000003 (ldr r0, [r11, #-12])
...	
r3	0x00000003 (mov r3, #3)
r11	0xFFFFEEFCC (Frame Pointer)
r13	0xFFFFEEFC0 (Stack Pointer)
r14	0xFF6677B4 (Link Register)

Stack Frame (ARM 32-Bit)

```
Dump of assembler code for function multiply_two:
=> 0x00010400 <+0>:    push    {r11}          ; (str r11, [sp, #-4]!)
    0x00010404 <+4>:    add     r11, sp, #0
    0x00010408 <+8>:    sub     sp, sp, #12
    0x0001040c <+12>:   str     r0, [r11, #-8]
    0x00010410 <+16>:   ldr     r3, [r11, #-8]
    0x00010414 <+20>:   lsl     r3, r3, #1
    0x00010418 <+24>:   mov     r0, r3
    0x0001041c <+28>:   add     sp, r11, #0
    0x00010420 <+32>:   pop     {r11}          ; (ldr r11, [sp], #4)
    0x00010424 <+36>:   bx      lr
End of assembler dump.
```

```
Dump of assembler code for function multiply_two:
    0x00010400 <+0>:    push    {r11}          ; (str r11, [sp, #-4]!)
    0x00010404 <+4>:    add     r11, sp, #0
    0x00010408 <+8>:    sub     sp, sp, #12
    0x0001040c <+12>:   str     r0, [r11, #-8]
    0x00010410 <+16>:   ldr     r3, [r11, #-8]
=> 0x00010414 <+20>:   lsl     r3, r3, #1
    0x00010418 <+24>:   mov     r0, r3
    0x0001041c <+28>:   add     sp, r11, #0
    0x00010420 <+32>:   pop     {r11}          ; (ldr r11, [sp], #4)
    0x00010424 <+36>:   bx      lr
End of assembler dump.
```

multiply_two 함수 호출

lr	
...	
lr	r11 + 0x04 == (main + 28)
r11	== r11
	r11 - 0x04
0x03	r11 - 0x08 (str r0, [r11, #-8])
	r11 - 0x0C == r13

함수 호출 후 Register

r0	0x00000003
...	
r3	0x00000003 (ldr r3, [r11, #-8])
r11	0xFFFFEEFBC
r13	0xFFFFEEFB0
r14	0x10444

Stack Frame (ARM 32-Bit)

multiply_two

lr	
...	
lr	r11 + 0x04 == (main + 28)
r11	== r11
	r11 - 0x04
0x03 << 1	r11 - 0x08 (lsl r3, r3, #1)
	r11 - 0x0C == r13

multiply_two 연산 수행 후 Register

r0	0x00000006 (mov r0, r3)
...	
r3	0x00000003
r11	0xFFFFEEFBC
r13	0xFFFFEEFB0
r14	0x10444

```
Dump of assembler code for function multiply_two:
0x00010400 <+0>:  push    {r11}           ; (str r11, [sp, #-4]!)
0x00010404 <+4>:  add     r11, sp, #0
0x00010408 <+8>:  sub     sp, sp, #12
0x0001040c <+12>: str     r0, [r11, #-8]
0x00010410 <+16>: ldr     r3, [r11, #-8]
0x00010414 <+20>: lsl     r3, r3, #1
0x00010418 <+24>: mov     r0, r3
=> 0x0001041c <+28>: add     sp, r11, #0
0x00010420 <+32>: pop     {r11}           ; (ldr r11, [sp], #4)
0x00010424 <+36>: bx      lr
End of assembler dump.
```

Stack Frame (ARM 32-Bit)

multiply_two (Before)

lr	
...	
lr	$r11 + 0x04 == (\text{main} + 28)$
r11	$== r11$
	$r11 - 0x04$
0x06	$r11 - 0x08 (\text{lsl } r3, r3, \#1)$
	$r11 - 0x0C == r13$

pop %rbp 후 Register

r0	0x00000006
...	
r3	0x00000003
r11	0xFFFFEEFCC
r13	0xFFFFEEFB0
r14	0x10444

multiply_two (After (add sp, r11, #0) && (pop %rbp))

lr	
r11	$== r11$
...	
lr	$r11 + 0x04 == (\text{main} + 28) == r13$

```
Dump of assembler code for function multiply_two:
0x00010400 <+0>:  push    {r11}           ; (str r11, [sp, #-4]!)
0x00010404 <+4>:  add     r11, sp, #0
0x00010408 <+8>:  sub     sp, sp, #12
0x0001040c <+12>: str     r0, [r11, #-8]
0x00010410 <+16>: ldr     r3, [r11, #-8]
0x00010414 <+20>: lsl     r3, r3, #1
0x00010418 <+24>: mov     r0, r3
0x0001041c <+28>: add     sp, r11, #0
0x00010420 <+32>: pop     {r11}           ; (ldr r11, [sp], #4)
=> 0x00010424 <+36>: bx      lr
End of assembler dump.
```


Stack Frame (ARM 32-Bit)

multiply_two (Before)

lr	
r11	== r11
...	
lr	bx lr == (main + 28) == r13

main 함수 복귀 (After bx lr)

lr	r11 + 0x04
r11	== r11
	r11 - 0x04
	r11 - 0x08
0x03	r11 - 0x0C == r13

Bx lr 후 Register

r0	0x00000006
...	
r3	0x00000003
r11	0xFFFFEEFCC
r13	0xFFFFEEFC0
r14	0x10444

Dump of assembler code for function main:

```

0x00010428 <+0>:  push    {r11, lr}
0x0001042c <+4>:  add     r11, sp, #4
0x00010430 <+8>:  sub     sp, sp, #8
0x00010434 <+12>: mov     r3, #3
0x00010438 <+16>: str     r3, [r11, #-12]
0x0001043c <+20>: ldr     r0, [r11, #-12]
0x00010440 <+24>: bl      0x10400 <multiply_two>
=> 0x00010444 <+28>: str     r0, [r11, #-8]
0x00010448 <+32>: ldr     r1, [r11, #-8]
0x0001044c <+36>: ldr     r0, [pc, #16] ; 0x10464 <main+60>
0x00010450 <+40>: bl      0x102e0 <printf@plt>
0x00010454 <+44>: mov     r3, #0
0x00010458 <+48>: mov     r0, r3
0x0001045c <+52>: sub     sp, r11, #4
0x00010460 <+56>: pop     {r11, pc}
0x00010464 <+60>: ldrdeq  r0, [r1], -r8
End of assembler dump.
    
```

Stack Frame (ARM 32-Bit)

main (Before)

lr	r11 + 0x04
r11	== r11
	r11 - 0x04
	r11 - 0x08
0x03	r11 - 0x0C == r13

Register

r0	0x00000006
...	
r3	0x00000003
r11	0xFFFFEEFCC
r13	0xFFFFEEFC0
r14	0x10448

Multiply_two 반환 값 전달(After str r0, [r11, #-8])

lr	r11 + 0x04
r11	== r11
	r11 - 0x04
0x06	r11 - 0x08 (str r0, [r11, #-8])
0x03	r11 - 0x0C == r13

```
Dump of assembler code for function main:
0x00010428 <+0>:    push    {r11, lr}
0x0001042c <+4>:    add     r11, sp, #4
0x00010430 <+8>:    sub     sp, sp, #8
0x00010434 <+12>:   mov     r3, #3
0x00010438 <+16>:   str     r3, [r11, #-12]
0x0001043c <+20>:   ldr     r0, [r11, #-12]
0x00010440 <+24>:   bl      0x10400 <multiply_two>
0x00010444 <+28>:   str     r0, [r11, #-8]
=> 0x00010448 <+32>:   ldr     r1, [r11, #-8]
0x0001044c <+36>:   ldr     r0, [pc, #16] ; 0x10464 <main+60>
0x00010450 <+40>:   bl      0x102e0 <printf@plt>
0x00010454 <+44>:   mov     r3, #0
0x00010458 <+48>:   mov     r0, r3
0x0001045c <+52>:   sub     sp, r11, #4
0x00010460 <+56>:   pop     {r11, pc}
0x00010464 <+60>:   ldrdeq  r0, [r1], -r8
End of assembler dump.
```


Stack Frame (ARM 64-Bit)

```
Dump of assembler code for function main:
=> 0x0000004000000784 <+0>: stp    x29, x30, [sp, #-32]!
0x0000004000000788 <+4>: mov    x29, sp
0x000000400000078c <+8>: mov    w0, #0x3                                     // #3
0x0000004000000790 <+12>: str    w0, [sp, #24]
0x0000004000000794 <+16>: ldr    w0, [sp, #24]
0x0000004000000798 <+20>: bl     0x400000076c <multiply_two>
0x000000400000079c <+24>: str    w0, [sp, #28]
0x00000040000007a0 <+28>: ldr    w1, [sp, #28]
0x00000040000007a4 <+32>: adrp   x0, 0x4000000000
0x00000040000007a8 <+36>: add    x0, x0, #0x860
0x00000040000007ac <+40>: bl     0x4000000650 <printf@plt>
0x00000040000007b0 <+44>: mov    w0, #0x0                                     // #0
0x00000040000007b4 <+48>: ldp    x29, x30, [sp], #32
0x00000040000007b8 <+52>: ret
End of assembler dump.
```

```
Dump of assembler code for function main:
0x0000004000000784 <+0>: stp    x29, x30, [sp, #-32]!
0x0000004000000788 <+4>: mov    x29, sp
0x000000400000078c <+8>: mov    w0, #0x3                                     // #3
0x0000004000000790 <+12>: str    w0, [sp, #24]
0x0000004000000794 <+16>: ldr    w0, [sp, #24]
=> 0x0000004000000798 <+20>: bl     0x400000076c <multiply_two>
0x000000400000079c <+24>: str    w0, [sp, #28]
0x00000040000007a0 <+28>: ldr    w1, [sp, #28]
0x00000040000007a4 <+32>: adrp   x0, 0x4000000000
0x00000040000007a8 <+36>: add    x0, x0, #0x860
0x00000040000007ac <+40>: bl     0x4000000650 <printf@plt>
0x00000040000007b0 <+44>: mov    w0, #0x0                                     // #0
0x00000040000007b4 <+48>: ldp    x29, x30, [sp], #32
0x00000040000007b8 <+52>: ret
End of assembler dump.
```

ARM 64-Bit 에서의 main 함수 호출

lr	== x30 (Return Address)
x29	
...	
0x3	sp + 0x18 (ldr w0, [sp, #24])
...	
	x29 - 0x20 == sp == x29

(stp x29, x30, [sp, #-32]!)

multiply_two 함수 호출 전 Register를 통한 num 인자 전달

w0	0x00000003 (mov w0, #0x3)
...	(str w0, [r11, #-12])
x29	0x4001811E10 (Frame Pointer)
x30	0x4001877E10 (Link Register)
sp	0x4001811E10 (Stack Frame)

Stack Frame (ARM 64-Bit)

ARM 64-Bit 에서의 main 함수 호출

lr	== x30 (Return Address)
x29	
...	
0x3	sp + 0x18 (ldr w0, [sp, #24])
...	
	x29 - 0x20 == sp == x29 (stp x29, x30, [sp, #-32]!)

multiply_two 함수 호출 전 Register를 통한 num 인자 전달

w0	0x00000003 (mov w0, #0x3)
...	(str w0, [r11, #-12])
x29	0x4001811E10 (Frame Pointer)
x30	0x4001877E10 (Link Register)
sp	0x4001811E10 (Stack Frame)

• ARM 32-Bit와 64-Bit의 Stack Frame 차이

- Register 와 Memory Bus Size가 64-Bit로 증가
- 범용 Register 의 개수가 증가 (r0 ~ r15 -> x0 ~ x30)
- Stack Pointer가 범용 Register를 사용하지 않음
- stp 명령어를 사용하여 FP와 LR, Stack 공간을 확보
- ldp, ret 명령어를 사용하여 Stack Frame 해제
- Stack Pointer를 이용하여 Stack Frame Offset 계산

조건문 (x64, ARM)

1. Jump 문

특정 조건 (Flag Register) 에 의해 분기하는 명령어, cmp 명령어를 통해 값을 비교하여 발생하는 Carry, Zero Flag 변화를 통해 분기하는 방법을 주로 이용한다.

2. Loop 문

특정 조건 (Flag Register) 에 의해 분기하여 반복하는 명령어, 반복할 때 마다 CX 값을 감소 시킴

3. x64 Flag Register(EFLAGS)

현재 연산 값의 상태 변화를 나타내는 Overflow, Sign, Zero, Auxiliary Carry, Parity, Carry Flag,

명령 수행 시 Register 값의 증감 방향을 나타내는 Direction,

시스템과 관련된 Interrupt, Trap, IOPL 등의 Flag를 지님

4. ARM Flag Register(Current Program Status Register)

x64와 유사한 현재 연산 값의 상태 변화를 나타내는 Overflow, Negative (Sign), Zero, Carry Flag,

시스템과 관련된 Interrupt Mask Field, 명령어와 관련된 Thumb Flag, 동작 모드를 설정하는 Mode Field 가 있다.

조건문 (x64)

```
#include "stdio.h"

int main ()
{
    int num = 10;

    for(int count = 0; count < num; count++)
    {
        if(count % 2 == 1)
            printf("%d", count);
        else
            printf(" ");
    }

    return 0;
}
```

0 ~ 9 까지의 정수를

홀수 값은 십진 정수로 출력

짝수 값은 공백 값을 출력

하는 예제

조건문 (x64)

```
Dump of assembler code for function main:
=> 0x000055555555169 <+0>:      endbr64
0x00005555555516d <+4>:      push    %rbp
0x00005555555516e <+5>:      mov     %rsp,%rbp
0x000055555555171 <+8>:      sub     $0x10,%rsp
0x000055555555175 <+12>:     movl    $0xa,-0x4(%rbp)
0x00005555555517c <+19>:     movl    $0x0,-0x8(%rbp)
0x000055555555183 <+26>:     jmp     0x555555551be <main+85>
0x000055555555185 <+28>:     mov     -0x8(%rbp),%eax
0x000055555555188 <+31>:     cld
0x000055555555189 <+32>:     shr     $0x1f,%edx
0x00005555555518c <+35>:     add     %edx,%eax
0x00005555555518e <+37>:     and     $0x1,%eax
0x000055555555191 <+40>:     sub     %edx,%eax
0x000055555555193 <+42>:     cmp     $0x1,%eax
0x000055555555196 <+45>:     jne     0x555555551b0 <main+71>
0x000055555555198 <+47>:     mov     -0x8(%rbp),%eax
0x00005555555519b <+50>:     mov     %eax,%esi
0x00005555555519d <+52>:     lea     0xe60(%rip),%rdi      # 0x555555556004
0x0000555555551a4 <+59>:     mov     $0x0,%eax
0x0000555555551a9 <+64>:     callq   0x55555555070 <printf@plt>
0x0000555555551ae <+69>:     jmp     0x555555551ba <main+81>
0x0000555555551b0 <+71>:     mov     $0x20,%edi
0x0000555555551b5 <+76>:     callq   0x55555555060 <putchar@plt>
0x0000555555551ba <+81>:     addl    $0x1,-0x8(%rbp)
0x0000555555551be <+85>:     mov     -0x8(%rbp),%eax
0x0000555555551c1 <+88>:     cmp     -0x4(%rbp),%eax
0x0000555555551c4 <+91>:     jl      0x55555555185 <main+28>
0x0000555555551c6 <+93>:     mov     $0x0,%eax
0x0000555555551cb <+98>:     leaveq  $0x0,%eax
0x0000555555551cc <+99>:     retq
End of assembler dump.
```

main 함수 실행 시

Stack Frame 을 할당하고, 이후 Stack Frame에 지역 변수인

num과 count 값을 각각 10과 0 값으로 초기화 한다.

count == 0은 짝수 이므로 즉시 printf 함수로 출력하고

이후 <main + 85>로 Jump 한다.

이때, count에 1을 더하고 count 의 값을 eax 에 대입하여,

eax 값을 num과 비교하여 num 값이 더 작을 때
jl (Jump Less) == ZF == 0 && CF == 1 일 경우 Jump

<main + 28>로 Jump 하여 Loop를 수행한다.

조건문 (x64)

```
Dump of assembler code for function main:
0x000055555555169 <+0>:    endbr64
0x00005555555516d <+4>:    push    %rbp
0x00005555555516e <+5>:    mov     %rsp,%rbp
0x000055555555171 <+8>:    sub     $0x10,%rsp
0x000055555555175 <+12>:   movl    $0xa,-0x4(%rbp)
0x00005555555517c <+19>:   movl    $0x0,-0x8(%rbp)
0x000055555555183 <+26>:   jmp     0x555555551be <main+85>
0x000055555555185 <+28>:   mov     -0x8(%rbp),%eax
0x000055555555188 <+31>:   cltd
0x000055555555189 <+32>:   shr     $0x1f,%edx
0x00005555555518c <+35>:   add     %edx,%eax
0x00005555555518e <+37>:   and     $0x1,%eax
0x000055555555191 <+40>:   sub     %edx,%eax
0x000055555555193 <+42>:   cmp     $0x1,%eax
0x000055555555196 <+45>:   jne     0x555555551b0 <main+71>
0x000055555555198 <+47>:   mov     -0x8(%rbp),%eax
0x00005555555519b <+50>:   mov     %eax,%esi
0x00005555555519d <+52>:   lea     0xe60(%rip),%rdi    # 0x555555556004
0x0000555555551a4 <+59>:   mov     $0x0,%eax
0x0000555555551a9 <+64>:   callq   0x55555555070 <printf@plt>
0x0000555555551ae <+69>:   jmp     0x555555551ba <main+81>
0x0000555555551b0 <+71>:   mov     $0x20,%edi
0x0000555555551b5 <+76>:   callq   0x55555555060 <putchar@plt>
0x0000555555551ba <+81>:   addl    $0x1,-0x8(%rbp)
0x0000555555551be <+85>:   mov     -0x8(%rbp),%eax
0x0000555555551c1 <+88>:   cmp     -0x4(%rbp),%eax
=> 0x0000555555551c4 <+91>:   jl      0x55555555185 <main+28>
0x0000555555551c6 <+93>:   mov     $0x0,%eax
0x0000555555551cb <+98>:   leaveq  %eax
0x0000555555551cc <+99>:   retq
End of assembler dump.
```

num == 1 부터는 cmp \$0x1, %eax 명령의 결과를 통해

jne (Jump if Not Equal), eax 값이 0x1 이 아닐 경우
(ZF 값이 0 인 경우)

<main +71>으로 분기, 0x1 인 경우<main +47>을 수행한다.

매 분기가 끝나면 <main +81>로 이동하여

eax (count)값을 num과 비교하여

count 값이 num 보다 작을 경우 <main + 28>로 분기하고

num 값이 더 작을 때 함수를 종료한다.

조건문 (ARM 32-Bit)

```
Dump of assembler code for function main:
=> 0x00010430 <+0>:  push    {r11, lr}
    0x00010434 <+4>:  add     r11, sp, #4
    0x00010438 <+8>:  sub     sp, sp, #8
    0x0001043c <+12>: mov     r3, #10
    0x00010440 <+16>: str     r3, [r11, #-8]
    0x00010444 <+20>: mov     r3, #0
    0x00010448 <+24>: str     r3, [r11, #-12]
    0x0001044c <+28>: b       0x1048c <main+92>
    0x00010450 <+32>: ldr     r3, [r11, #-12]
    0x00010454 <+36>: cmp     r3, #0
    0x00010458 <+40>: and     r3, r3, #1
    0x0001045c <+44>: rsblt   r3, r3, #0
    0x00010460 <+48>: cmp     r3, #1
    0x00010464 <+52>: bne     0x10478 <main+72>
    0x00010468 <+56>: ldr     r1, [r11, #-12]
    0x0001046c <+60>: ldr     r0, [pc, #56] ; 0x104ac <main+124>
    0x00010470 <+64>: bl      0x10304 <printf@plt>
    0x00010474 <+68>: b       0x10480 <main+80>
    0x00010478 <+72>: mov     r0, #32
    0x0001047c <+76>: bl      0x10328 <putchar@plt>
    0x00010480 <+80>: ldr     r3, [r11, #-12]
    0x00010484 <+84>: add     r3, r3, #1
    0x00010488 <+88>: str     r3, [r11, #-12]
    0x0001048c <+92>: ldr     r2, [r11, #-12]
    0x00010490 <+96>: ldr     r3, [r11, #-8]
    0x00010494 <+100>: cmp     r2, r3
    0x00010498 <+104>: blt     0x10450 <main+32>
    0x0001049c <+108>: mov     r3, #0
    0x000104a0 <+112>: mov     r0, r3
    0x000104a4 <+116>: sub     sp, r11, #4
    0x000104a8 <+120>: pop     {r11, pc}
    0x000104ac <+124>: andeq   r0, r1, r0, lsr #10
End of assembler dump.
```

동작 자체는 x64와 유사하나 Branch (b) 명령어를 사용한다.

main 함수 시 Stack Frame 을 할당하고,

지역 변수 num, count를 각각 10과 0으로 초기화 한다.

이후 **<main + 92>** 로 분기하여 count 값을 증가 시키고

count 값을 r2, num 값을 r3에 불러와 두 값을 cmp 명령을 수행하여

blt (Branch if Less Than) 에 따라 count 값이 num 값 보다 작으면

<main + 32>로 분기 한다.

조건문 (ARM 32-Bit)

```
Dump of assembler code for function main:
0x00010430 <+0>:  push    {r11, lr}
0x00010434 <+4>:  add     r11, sp, #4
0x00010438 <+8>:  sub     sp, sp, #8
0x0001043c <+12>:  mov     r3, #10
0x00010440 <+16>:  str     r3, [r11, #-8]
0x00010444 <+20>:  mov     r3, #0
0x00010448 <+24>:  str     r3, [r11, #-12]
0x0001044c <+28>:  b       0x1048c <main+92>
0x00010450 <+32>:  ldr     r3, [r11, #-12]
0x00010454 <+36>:  cmp     r3, #0
0x00010458 <+40>:  and     r3, r3, #1
0x0001045c <+44>:  rsblt   r3, r3, #0
0x00010460 <+48>:  cmp     r3, #1
0x00010464 <+52>:  bne     0x10478 <main+72>
0x00010468 <+56>:  ldr     r1, [r11, #-12]
0x0001046c <+60>:  ldr     r0, [pc, #56] ; 0x104ac <main+124>
0x00010470 <+64>:  bl      0x10304 <printf@plt>
0x00010474 <+68>:  b       0x10480 <main+80>
0x00010478 <+72>:  mov     r0, #32
0x0001047c <+76>:  bl      0x10328 <putchar@plt>
0x00010480 <+80>:  ldr     r3, [r11, #-12]
0x00010484 <+84>:  add     r3, r3, #1
0x00010488 <+88>:  str     r3, [r11, #-12]
0x0001048c <+92>:  ldr     r2, [r11, #-12]
0x00010490 <+96>:  ldr     r3, [r11, #-8]
0x00010494 <+100>:  cmp     r2, r3
=> 0x00010498 <+104>:  blt     0x10450 <main+32>
0x0001049c <+108>:  mov     r3, #0
0x000104a0 <+112>:  mov     r0, r3
0x000104a4 <+116>:  sub     sp, r11, #4
0x000104a8 <+120>:  pop     {r11, pc}
0x000104ac <+124>:  andeq   r0, r1, r0, lsr #10
End of assembler dump.
```

이후 cmp 명령어를 통해 count (r3)와 즉시 값 1을 비교하고

bne (Branch if Not Equal) 명령어를 통해

count 값이 1이 아닐 경우 <main + 72>로 분기하고

1일 경우 <main + 56> 을 수행한다.

매 분기가 끝나면 <main +80>로 이동하여

r2 (count)값을 r3 (num)과 비교하여

count 값이 num 보다 작을 경우 <main + 32>로 분기하고

num 값이 더 작을 때 함수를 종료한다.

ARM과 x64(x86) Assembly 차이점

1. Register 간 Data 교환이 가능한 x64와 달리 ARM은 Store / Load 구조로 Memory를 통하여 접근해야 한다.
2. 가변 길이 기계어를 갖는 x64와 달리 ARM 명령어는 Bus Size에 맞추어 정렬되어 있다.
3. Stack Frame 해제 시 RIP 하나만을 사용하는 x64와 달리 ARM은 Link Register와 Program Counter를 나누어 사용한다.
4. 문법에 맞춰 Operand의 방향이 한 쪽으로 결정되는 Intel Assembly와 달리 ARM은 Register와 Memory의 상관 관계, 명령어의 구조에 따라 방향이 결정된다.