Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 및 회로 설계 전문가 과정

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 장성환
redmk1025@gmail.com

CISC RISC 의 핵심적 차이점은 **??? ARM** 은 **load/store** 아키텍쳐 **!**
mov, add, lsl, asr, mrs, mul, mla, umull, umlal, ldr, ldreqb, strb, !, stm, stmia, ldmia, subfunc 기법

func.c

```
#include <stdio.h>

int my_func(int num){
   return num*2;
}

int main(void){
   int res, num =2;
   res = my_func(num);
   printf("res = %d\n", res);
   return 0;
}
```

tar rem localhost:1234 으로 실행
b *addr 로 원하는 위치 주소값 위치 (si 로 함수이동 가능케 함. ni 는 코드별 이동)
p $r1 / p/x $r2 같이 레지스터값 확인

r11 이 BP 의 역할
lr 이 PC (복귀 주소)의 역할을 한다.
함수의 리턴값은 무조건 r0 이다. (따라서 함수 호출 후에 r0 값을 보면 이상한 값이 나오게 된다. 또한 되도록 인자는 r4, r5, r6, r7 를 사용하는게 좋다.)

r7 은 시스템 콜을 할때 주로 사용한다.

<메인 asm>

```
(gdb) disas
Dump of assembler code for function main:
   0x00010460 <+0>:     push    {r11, lr}
   0x00010464 <+4>:     add     r11, sp, #4
   0x00010468 <+8>:     sub     sp, sp, #8
=> 0x0001046c <+12>:    mov     r3, #2
   0x00010470 <+16>:    str     r3, [r11, #-12]
   0x00010474 <+20>:    ldr     r0, [r11, #-12]
   0x00010478 <+24>:    bl      0x10438 <my_func>
   0x0001047c <+28>:    str     r0, [r11, #-8]
   0x00010480 <+32>:    ldr     r1, [r11, #-8]
   0x00010484 <+36>:    ldr     r0, [pc, #16]   ; 0x1049c <main+60>
   0x00010488 <+40>:    bl      0x102e0 <printf@plt>
   0x0001048c <+44>:    mov     r3, #0
   0x00010490 <+48>:    mov     r0, r3
   0x00010494 <+52>:    sub     sp, r11, #4
   0x00010498 <+56>:    pop     {r11, pc}
   0x0001049c <+60>:    andeq   r0, r1, r0, lsl r5
End of assembler dump.
(gdb)
```

<sub func asm>

```
(gdb) disas
Dump of assembler code for function my_func:
=> 0x00010438 <+0>:     push    {r11}           ; (str r11, [sp, #-4]!)
   0x0001043c <+4>:     add     r11, sp, #0
   0x00010440 <+8>:     sub     sp, sp, #12
   0x00010444 <+12>:    str     r0, [r11, #-8]
   0x00010448 <+16>:    ldr     r3, [r11, #-8]
   0x0001044c <+20>:    lsl     r3, r3, #1
   0x00010450 <+24>:    mov     r0, r3
   0x00010454 <+28>:    sub     sp, r11, #0
   0x00010458 <+32>:    pop     {r11}           ; (ldr r11, [sp], #4)
   0x0001045c <+36>:    bx      lr
End of assembler dump.
```

asm 분석
sp 를 F8 이라고 가정

push {r11, lr}
sp 에 lr 저장, sp-- 다시 r11 저장 후 sp-- // sp = F0

add r11, sp, #4
r11 = sp + 4 = F4 (base point)

sub sp, sp, #8
sp = sp - 8 = E8 (스택 공간 생성)

mov r3, #2
r3 = 2

str r3 [r11, #-12]
*(r11-12) = r3 -> *E8 = 2

ldr r0 [r11, #-12]
r0 = *E8 = 2

bl 0x10438 <my_func>
blanch link 로서 복귀주소 lr = 0x1047C 이며 다음 실행은 0x10438 로 이동

push {r11}
sp 에 r11 저장 즉, *E8 = F4, sp--로 인해 sp=E4

add r11, sp, #0
r11 = E4 (두 번째 BP 설정)
..... 이하 같은 방식 동작.

func2.c

```
#include <stdio.h>

int my_func(int n1, int n2, int n3, int n4, int n5){
   return n1 + n2 + n3 + n4 + n5;
}
int main(void){
   int res, n1=2, n2=3, n3=4, n4=5, n5=6;
   res = my_func(n1,n2,n3,n4,n5);
   printf("res = %d\n", res);
   return 0;
}
```

<sub func asm>

```
(gdb) disas
Dump of assembler code for function my_func:
=> 0x00010438 <+0>:     push    {r11}           ; (str r11, [sp, #-4]!)
   0x0001043c <+4>:     add     r11, sp, #0
   0x00010440 <+8>:     sub     sp, sp, #20
   0x00010444 <+12>:    str     r0, [r11, #-8]
   0x00010448 <+16>:    str     r1, [r11, #-12]
   0x0001044c <+20>:    str     r2, [r11, #-16]
   0x00010450 <+24>:    str     r3, [r11, #-20] ; 0xffffffec
   0x00010454 <+28>:    ldr     r2, [r11, #-8]
   0x00010458 <+32>:    ldr     r3, [r11, #-12]
   0x0001045c <+36>:    add     r2, r2, r3
   0x00010460 <+40>:    ldr     r3, [r11, #-16]
   0x00010464 <+44>:    add     r2, r2, r3
   0x00010468 <+48>:    ldr     r3, [r11, #-20] ; 0xffffffec
   0x0001046c <+52>:    add     r2, r2, r3
   0x00010470 <+56>:    ldr     r3, [r11, #4]
   0x00010474 <+60>:    add     r3, r2, r3
   0x00010478 <+64>:    mov     r0, r3
   0x0001047c <+68>:    sub     sp, r11, #0
   0x00010480 <+72>:    pop     {r11}           ; (ldr r11, [sp], #4)
   0x00010484 <+76>:    bx      lr
End of assembler dump.
(gdb)
```

<main asm>

```
(gdb) disas
Dump of assembler code for function main:
   0x00010488 <+0>:     push    {r11, lr}
   0x0001048c <+4>:     add     r11, sp, #4
   0x00010490 <+8>:     sub     sp, sp, #32
=> 0x00010494 <+12>:    mov     r3, #2
   0x00010498 <+16>:    str     r3, [r11, #-28] ; 0xffffffe4
   0x0001049c <+20>:    mov     r3, #3
   0x000104a0 <+24>:    str     r3, [r11, #-24] ; 0xffffffe8
   0x000104a4 <+28>:    mov     r3, #4
   0x000104a8 <+32>:    str     r3, [r11, #-20] ; 0xffffffec
   0x000104ac <+36>:    mov     r3, #5
   0x000104b0 <+40>:    str     r3, [r11, #-16]
   0x000104b4 <+44>:    mov     r3, #6
   0x000104b8 <+48>:    str     r3, [r11, #-12]
   0x000104bc <+52>:    ldr     r3, [r11, #-12]
   0x000104c0 <+56>:    str     r3, [sp]
   0x000104c4 <+60>:    ldr     r3, [r11, #-16]
   0x000104c8 <+64>:    ldr     r2, [r11, #-20] ; 0xffffffec
   0x000104cc <+68>:    ldr     r1, [r11, #-24] ; 0xffffffe8
   0x000104d0 <+72>:    ldr     r0, [r11, #-28] ; 0xffffffe4
   0x000104d4 <+76>:    bl      0x10438 <my_func>
   0x000104d8 <+80>:    str     r0, [r11, #-8]
   0x000104dc <+84>:    ldr     r1, [r11, #-8]
   0x000104e0 <+88>:    ldr     r0, [pc, #16]   ; 0x104f8 <main+112>
   0x000104e4 <+92>:    bl      0x102e0 <printf@plt>
   0x000104e8 <+96>:    mov     r3, #0
   0x000104ec <+100>:   mov     r0, r3
   0x000104f0 <+104>:   sub     sp, r11, #4
   0x000104f4 <+108>:   pop     {r11, pc}
   0x000104f8 <+112>:   andeq   r0, r1, r12, ror #10
End of assembler dump.
(gdb)
```

ARM 에서 함수에 전달은 레지스터를 4 개만 쓴다. 속도를 위해서 인자를 4 개까지 쓰는게 좋다. (어쩔 수 없는 경우 구조체를 사용하면 인자 갯수를 늘릴 수 있다.)

func3.c

```
#include <stdio.h>

int my_func(int n1, int n2, int n3, int n4, int n5, int n6, int n7){
   return n1 + n2 + n3 + n4 + n5+ n6 + n7;
}
int main(void){
   int res, n1=2, n2=3, n3=4, n4=5, n5=6, n6=7, n7 =8;
   res = my_func(n1,n2,n3,n4,n5,n6,n7);
   printf("res = %d\n", res);
   return 0;
}
```

<sub func asm>

```
Dump of assembler code for function my_func:
=> 0x00010438 <+0>:     push    {r11}           ; (str r11, [sp, #-4]!)
   0x0001043c <+4>:     add     r11, sp, #0
   0x00010440 <+8>:     sub     sp, sp, #20
   0x00010444 <+12>:    str     r0, [r11, #-8]
   0x00010448 <+16>:    str     r1, [r11, #-12]
   0x0001044c <+20>:    str     r2, [r11, #-16]
   0x00010450 <+24>:    str     r3, [r11, #-20] ; 0xffffffec
   0x00010454 <+28>:    ldr     r2, [r11, #-8]
   0x00010458 <+32>:    ldr     r3, [r11, #-12]
   0x0001045c <+36>:    add     r2, r2, r3
   0x00010460 <+40>:    ldr     r3, [r11, #-16]
   0x00010464 <+44>:    add     r2, r2, r3
   0x00010468 <+48>:    ldr     r3, [r11, #-20] ; 0xffffffec
   0x0001046c <+52>:    add     r2, r2, r3
   0x00010470 <+56>:    ldr     r3, [r11, #4]
   0x00010474 <+60>:    add     r2, r2, r3
   0x00010478 <+64>:    ldr     r3, [r11, #8]
   0x0001047c <+68>:    add     r2, r2, r3
   0x00010480 <+72>:    ldr     r3, [r11, #12]
   0x00010484 <+76>:    add     r3, r2, r3
   0x00010488 <+80>:    mov     r0, r3
   0x0001048c <+84>:    sub     sp, r11, #0
   0x00010490 <+88>:    pop     {r11}           ; (ldr r11, [sp], #4)
   0x00010494 <+92>:    bx      lr
```

<main asm>

```
   0x00010498 <+0>:     push    {r11, lr}
   0x0001049c <+4>:     add     r11, sp, #4
   0x000104a0 <+8>:     sub     sp, sp, #48     ; 0x30
=> 0x000104a4 <+12>:    mov     r3, #2
   0x000104a8 <+16>:    str     r3, [r11, #-36] ; 0xffffffdc
   0x000104ac <+20>:    mov     r3, #3
   0x000104b0 <+24>:    str     r3, [r11, #-32] ; 0xffffffe0
   0x000104b4 <+28>:    mov     r3, #4
   0x000104b8 <+32>:    str     r3, [r11, #-28] ; 0xffffffe4
   0x000104bc <+36>:    mov     r3, #5
   0x000104c0 <+40>:    str     r3, [r11, #-24] ; 0xffffffe8
   0x000104c4 <+44>:    mov     r3, #6
   0x000104c8 <+48>:    str     r3, [r11, #-20] ; 0xffffffec
   0x000104cc <+52>:    mov     r3, #7
   0x000104d0 <+56>:    str     r3, [r11, #-16]
   0x000104d4 <+60>:    mov     r3, #8
   0x000104d8 <+64>:    str     r3, [r11, #-12]
   0x000104dc <+68>:    ldr     r3, [r11, #-12]
   0x000104e0 <+72>:    str     r3, [sp, #8]
   0x000104e4 <+76>:    ldr     r3, [r11, #-16]
   0x000104e8 <+80>:    str     r3, [sp, #4]
   0x000104ec <+84>:    ldr     r3, [r11, #-20] ; 0xffffffec
   0x000104f0 <+88>:    str     r3, [sp]
   0x000104f4 <+92>:    ldr     r3, [r11, #-24] ; 0xffffffe8
   0x000104f8 <+96>:    ldr     r2, [r11, #-28] ; 0xffffffe4
   0x000104fc <+100>:   ldr     r1, [r11, #-32] ; 0xffffffe0
   0x00010500 <+104>:   ldr     r0, [r11, #-36] ; 0xffffffdc
   0x00010504 <+108>:   bl      0x10438 <my_func>
   0x00010508 <+112>:   str     r0, [r11, #-8]
   0x0001050c <+116>:   ldr     r1, [r11, #-8]
   0x00010510 <+120>:   ldr     r0, [pc, #16]   ; 0x10528 <main+144>
   0x00010514 <+124>:   bl      0x102e0 <printf@plt>
   0x00010518 <+128>:   mov     r3, #0
   0x0001051c <+132>:   mov     r0, r3
   0x00010520 <+136>:   sub     sp, r11, #4
   0x00010524 <+140>:   pop     {r11, pc}
   0x00010528 <+144>:   muleq   r1, r12, r5
```

stack 자체를 변수 값 뿐만 아니라 함수에 전달하는 (4 개가 초과하는) 변수에 대하여도 저장 공간을 만든다. 함수에서 레지스터로 4 개를 전달받고, 나머지 부분은 스택에서 불러와서 인자로 활용한다.