

임베디드시스템 설계 및 실습 - 보고서

제출일: 2021-04-28; Lab 3

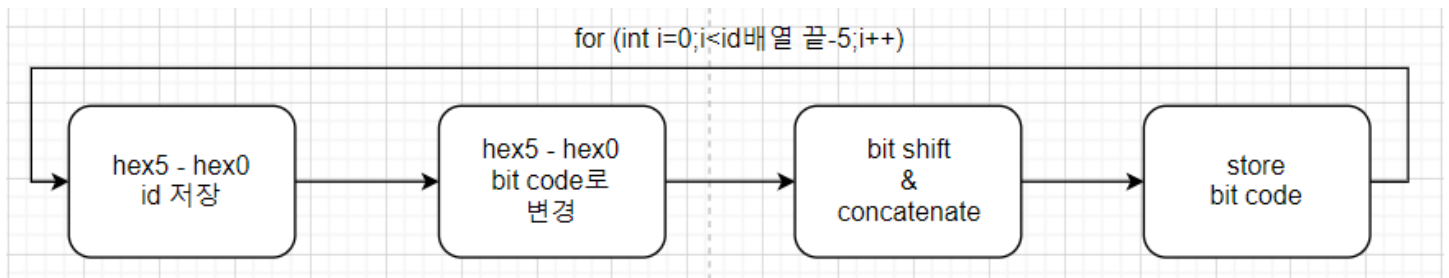
이름: 박관영(2016124099),김서영(2017106007),오한별(2018124120) ; 4조

✓ Assignment 1

개요

본 lab에서는 Linux 기반으로 동작하는 HPS에서 virtual address pointer를 통해 FPGA의 peripheral을 MMIO 형태로 제어하는 방법을 학습한다. Light Weight HPS-to-FPGA bridge를 이용하여 HPS의 master port와 FPGA fabric의 memory mapped slave port를 연결해준다. Slave로 DE1-SoC 보드 상의 HEX를 사용하고, HEX에 학생들의 학번을 banner scroll 형태로 display한다. Interrupt I/O는 사용하지 않으며 MMIO의 read/write만을 사용하여 학번이 display되도록 한다. 학번은 int형 배열 id에 한 자리 씩 저장하며, for문으로 배열의 끝까지 읽어 한 자리 씩 display할 수 있도록 한다.

Seven segment로 display하기 위해 bit code를 생성하고, 생성한 bit code를 해당 memory에 mapping한다. 이 때 HEX3-HEX0, HEX5-HEX4는 각각 하나의 주소에 mapping 되어있으므로, bit shift를 통해 알맞은 자리에 bit code가 저장될 수 있도록 한다. 프로그램의 flow는 다음과 같다.



진행 내용

```
#include<stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "address_map_arm.h"

//비트코드 생성을 위한 int 형 배열
int BIT_CODE[10]={0b00111111, 0b00000110, 0b01011011, 0b01001111, 0b01100110,
0b01101101, 0b01111101, 0b00000111, 0b01111111, 0b01100111};

//학생들의 학번을 한 자리 씩 저장한 배열
int id[30] = {2,0,1,6,1,2,4,0,9,9,2,0,1,7,1,0,6,0,0,7,2,0,1,8,1,2,4,1,2,0};

//숫자를 HEX에 나타낼 수 있도록 비트코드로 변환하는 함수
int disp(int number){
    int bitcode=0;
    bitcode = BIT_CODE[number];
```

임베디드시스템 설계 및 실습 - 보고서

```
    return bitcode;
}

int main(void){
    int fd=0, i=0;
    void *lw_virtual;
    volatile int *hex3_hex0;
    volatile int *hex5_hex4;
    int hex5=0, hex4=0, hex3=0, hex2=0, hex1=0, hex0=0;

    fd = open("/dev/mem",(O_RDWR | O_SYNC));
    lw_virtual = mmap(NULL, LW_BRIDGE_SPAN, (PROT_READ|PROT_WRITE),MAP_SHARED,fd,LW_BRIDGE_BASE);

    for(i=0;i<25;i++){
        //id 배열을 읽어 hex의 위치를 나타내는 변수 hex5-hex0에 저장
        hex5= id[i]; hex4=id[i+1]; hex3=id[i+2]; hex2=id[i+3]; hex1=id[i+4]; hex0=id[i+5];

        //숫자를 해당 비트코드로 바꿔줌
        hex5=disp(hex5); hex4=disp(hex4); hex3=disp(hex3); hex2=disp(hex2); hex1=disp(hex1); hex0=disp(hex0);

        //hex3-hex0, hex5-hex4 는 각각 하나의 Memory에 mapping되므로 bit code를 다음과 같이 shift하여 나타냄
        hex3=hex3<<24; hex2=hex2<<16; hex1=hex1<<8;
        hex5=hex5<<8;

        //최종적으로 생성된 bit code
        hex0=hex3+hex2+hex1+hex0;
        hex4=hex5+hex4;

        //생성한 bitcode를 해당 memory에 저장
        *hex3_hex0=hex0;
        *hex5_hex4=hex4;

        //delay
        usleep(500000);
    }
    //memory해제
    munmap(lw_virtual, LW_BRIDGE_BASE);
    close(fd);
    return 0;
}
```

1) fd = open("/dev/mem",(O_RDWR | O_SYNC));

File을 열 때 다음과 같이 flag로 모드를 설정할 수 있다. 본 설계에서는 O_RDWR로 파일을 읽기/쓰기 용으로 열고, or 연산자 (|)를 사용해 O_SYNC도 가능하게 한다. O_SYNC는 파일의 수정 시간 속성도 파일 수정을 완료할 때까지 기다린다. O_SYNC옵션을 설정하면 프로그램의 실행 속도는 느려질 수 있으나 디스크에 확실하게 저장됨을 보장한다.

2) lw_virtual = mmap(NULL, LW_BRIDGE_SPAN, (PROT_READ|PROT_WRITE),MAP_SHARED,fd,LW_BRIDGE_BASE);

임베디드시스템 설계 및 실습 - 보고서

mmap함수는 파일기술폰자 fd가 가리키는 객체를 파일에서 offset바이트 지점을 기준으로 len 바이트만큼 memory에 mapping하도록 커널에 요청한다. 접근 권한은 prot에 지정하고, 추가적인 동작 방식은 flag에 지정한다. 본 실습에서는 len = LW_BRIDGE_SPAN 이고 read/write가 가능하도록 접근 권한을 설정한다. 추가적으로 MAP_SHARED flag를 지정하고 offset은 LW_BRIDGE_BASE로 지정한다.

```
COM9 - PuTTY
GNU nano 2.2.6 File: banner.c

int disp(int number){
    int bitcode=0;
    bitcode=BIT_CODE[number];
    return bitcode;
}

int main(void){
    int fd=0, i=0;
    void *lw_virtual;
    volatile int *hex3_hex0;
    volatile int *hex5_hex4;
    // volatile int *ledr;

    fd = open("/dev/mem", (O_RDWR | O_SYNC));
    lw_virtual = mmap(NULL, LW_BRIDGE_SPAN, (PROT_READ|PROT_WRITE), MAP_SHARED, fd, 0);

    int hex5=0, hex4=0, hex3=0, hex2=0, hex1=0, hex0=0;
    int led=0;

    ^G Get Help    ^O WriteOut    ^R Read File    ^Y Prev Page    ^K Cut Text     ^C Cur Pos
    ^X Exit        ^J Justify     ^W Where Is    ^V Next Page    ^U UnCut Text  ^T To Spell
```

Nano editor를 사용하여 코드를 편집하였다.

진행 결과

1. Host system에 있는 Address_map_arm.h 파일을 DE1-Soc보드상으로 옮긴다.

```
root@delsoclinux:~# cd fat_partition
root@delsoclinux:~/fat_partition# ls -l
total 6508
drwxr-xr-x 2 root root    4096 Jun 13  2018 System Volume Information
-rwxr-xr-x 1 root root    3064 Jan 22  2016 address_map_arm.h
-rwxr-xr-x 1 root root 2316870 Jul 17  2015 soc_system.rbf
-rwxr-xr-x 1 root root   19147 Jul 14  2016 socfpga.dtb
-rwxr-xr-x 1 root root 4315040 Aug  8  2016 uImage
```

→ host system에서 sd카드 리더기로 sd카드 메모리상으로 address_map_arm.h 파일을 복사한다. fat_partition 디렉토리 상에 파일이 있음을 확인할 수 있다.

2. 실행할 파일이 있는 디렉토리에 address_map_arm.h 파일을 옮김

```
root@delsoclinux:~# cd fat_partition
root@delsoclinux:~/fat_partition# cp address_map_arm.h ~/lab3/
```

임베디드시스템 설계 및 실습 - 보고서

→ cp 명령어로 옮길 파일과 해당 디렉토리를 지정한다.

3. c파일 컴파일

```
root@delsoclinux:~/lab3# gcc -c banner.c
```

→ cp 명령어로 옮길 파일과 해당 디렉토리를 지정한다.

4. object file 생성

```
root@delsoclinux:~/lab3# gcc banner.o -o id_banner
```

→ id_banner로 파일명을 변경한다.

```
root@delsoclinux:~/lab3# ls
address_map_arm.h  banner.c  banner.o  id_banner
```

5. 실행

```
root@delsoclinux:~/lab3# ./id_banner
```

출력 순서는 2016124099 -> 2017106007 -> 2018124120 이고 hex5-hex0에 6개씩 display하면 다음과 같다.

201612	016124	161240	612409	124099
240992	409920	099201	920171	201710
017106	171060	7110600	106007	060072

임베디드시스템 설계 및 실습 - 보고서

				
600720	007201	072018	710600	106007
				
060072	600720	007201	072018	720181
				
201812	018124	181241	812412	124120
				

➔ 순차적으로 HEX5-HEX0에 display됨을 확인할 수 있다.

결과 분석 및 팀원 간 토의 사항

1) HPS to FPGA Bridge를 사용하기 위해 sys/mman.h를 include한다. mmap함수를 사용하며 함수 원형은 다음과 같다.

Void *mmap(void *addr, size_T,len, int prot, int flags, int fd, off_t, offset);

* return값 : mapping이 시작하는 실제 메모리 주소

Addr	커널에게 파일을 어디에 mapping하면 좋을지 제안하는 값으로 본 설계에서는 NULL을 사용하였다.
Len	mapping시킬 메모리 영역의 길이
Prot	Mapping에 원하는 메모리 보호 정책

임베디드시스템 설계 및 실습 - 보고서

	-PROT_READ : 읽기 가능한 페이지 -PROT_WRITE : 쓰기 가능한 페이지 -PROT_EXEC : 실행 가능한 페이지 -PROT_NONE : 접근할 수 없는 페이지
Flag	Mapping 유형과 동작 구성 요소 -MAP_FIXED : addr과 len 매개변수가 기존의 mapping과 겹칠 경우 중첩 page를 버리고 새 mapping으로 대체 -MAP_SHARED : 동일 파일을 mapping한 모든 프로세스들이 공유, 객체 접근에 대한 동기화를 위해 msync, munmap을 사용함 -MAP_PRIVATE : mapping을 공유하지 않아 파일은 쓰기 후 복사로 mapping되며 변경된 메모리 속성은 실제 파일에는 반영되지 않음
Fd	파일 기술자
offset	Mapping할 때 len의 시작점을 지정함

디자인하는 시스템에 따라 설정을 변경하여 사용할 수 있다.

2) 잘못된 memory를 지정하면 compile과 object file 생성은 할 수 있으나, 실행 시킬 경우 segmentation fault error가 발생한다. 이를 gdb debugger를 사용하여 확인하면 다음과 같다.

```

root@delsoclinux:~/lab3# gdb ./id_banner
GNU gdb (Ubuntu/Linaro 7.4-2012.02-0ubuntu2) 7.4-2012.02
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabi".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/root/lab3/id_banner...(no debugging symbols found)...
done.
(gdb) run
Starting program: /home/root/lab3/id_banner

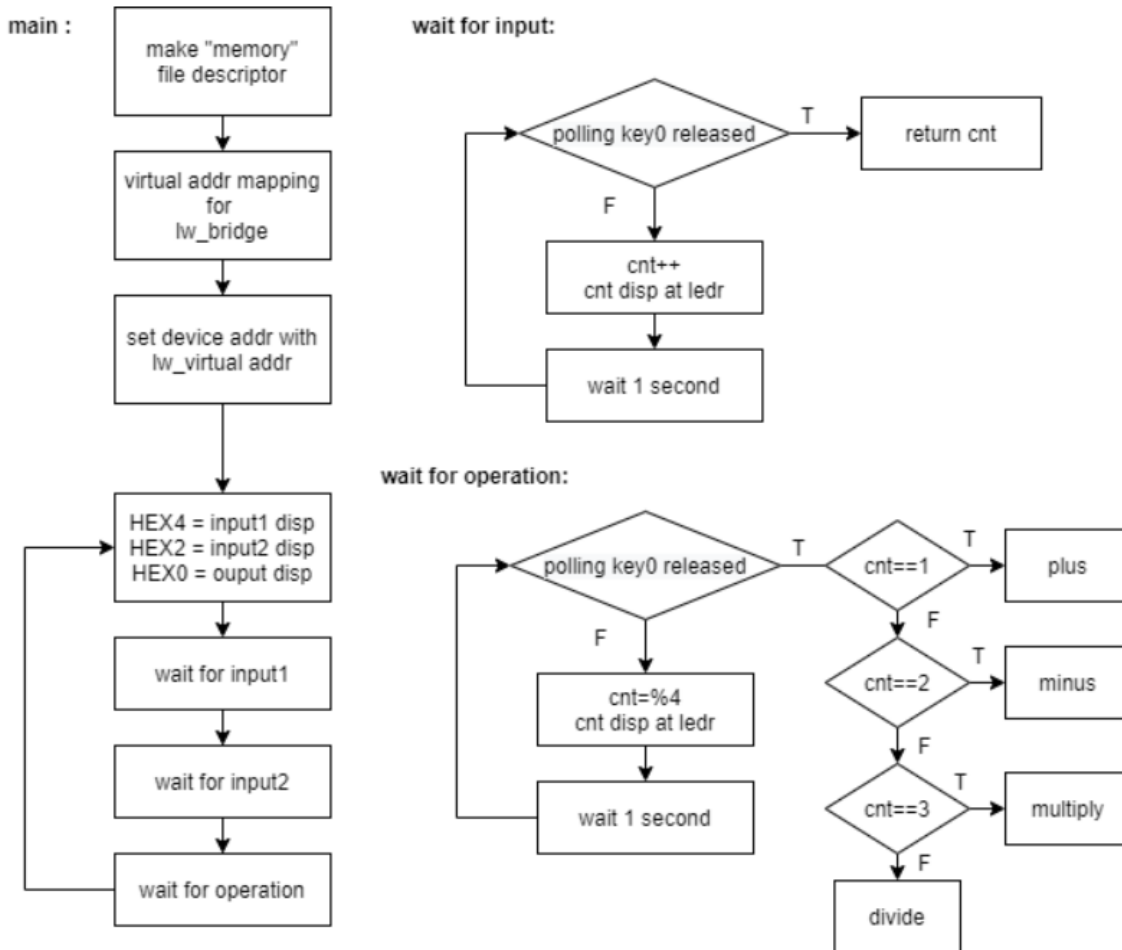
Program received signal SIGSEGV, Segmentation fault.
0x000085f2 in main ()
  
```

Segmentation fault는 컴퓨터 sw 실행 중에 일어날 수 있는 특수한 오류이며, 허용되지 않은 메모리 영역에 접근을 시도할 경우 발생한다. Linux는 unix 계열의 운영체제이므로 잘못된 메모리에 접근하는 프로세스는 SIGSEGV 신호를 받는다.

임베디드시스템 설계 및 실습 - 보고서

✓ Assignment 2

개요



본 lab에서는 linux-program을 MMIO(Memory-Mapped IO) 형태로 구현하여 DE1-SoC의 device를 control하는 것을 목표로 한다. MMIO란, CPU가 peripheral에 access하는 것을 load/store instruction을 사용하여 memory access와 같은 방식으로 수행하는 것을 말한다. 즉, peripheral의 address가 memory 영역 상에 존재하여 해당 address에 load/store를 통하여 접근하는 것이 해당 peripheral에 값을 입력하고, 읽어오는 것과 같이 동작하게 되는 것이다.

Cyclone V SoC platform의 Architecture는 HPS part와 FPGA part로 이루어져 있는데, HPS part에서 linux program이 동작하게 되고 control 하고자 하는 device는 FPGA part에 있기 때문에 FPGA Bridge에 access한 후 원하는 peripheral에 access하는 형태로 program을 구현하게 된다. 이 때 Bridge에서는 AXI(Advanced eXtensible Interface) protocol을 사용하게 된다.

본 lab에서는 MMIO 형태로 device를 control하기 때문에 memory device의 file descriptor를 open하여 사용하게 되고, Lightweight bridge의 base address를 통해 접근한 각 peripheral들은 polled IO 형태로 control하게 된다. 즉, 각 peripheral의 상태를 매번 확인하면서 program을 진행하게 된다. 또한, compiler에서 최적화를 통한 오류를 막기 위해 C에서 각 peripheral의 주소를 나타내는 pointer 변수를 선언할 때에는 volatile keyword를 사용하여 선언한다.

진행 내용

```
#include <stdio.h>
#include <unistd.h>
```

임베디드시스템 설계 및 실습 - 보고서

```
#include <fcntl.h>
#include <sys/mman.h> // mmap, unmmap 함수를 사용하기 위한 헤더파일.
#include "address_map_arm.h" // arm architecture의 peripherals의 주소가 지정되어 있는 헤더파일.

int segment(int a){ // one digit positive integer를 segment code로 바꾸는 함수.
    int b= (a<=0)? 0x3f:W
        (a==1)? 0x06:W
        (a==2)? 0x5b:W
        (a==3)? 0x4f:W
        (a==4)? 0x66:W
        (a==5)? 0x6d:W
        (a==6)? 0x7d:W
        (a==7)? 0x07:W
        (a==8)? 0x7f: 0x6f;
    return b;
}

int plus(int input1, int input2){ // plus 연산.
    int tmp=0;
    tmp = input1 + input2;
    return tmp;
}

int minus(int input1, int input2){ // minus 연산.
    int tmp=0;
    tmp = input1 - input2;
    return tmp;
}

int mul(int input1, int input2){ // multiply 연산.
    int tmp=0;
    tmp = input1 * input2;
    return tmp;
}

int div(int input1, int input2){ // divide 연산.
    int tmp =0;
    tmp = input1 / input2;
    return tmp;
}

//// 한자리 positive integer를 하나의 key를 통해 입력 받게 된다.
int wait_for_value(volatile int* key, volatile int* led_r){
    int cnt=0;
    while(1){ // key0가 released되는 순간을 polling.
        if(((key)&0x1)==0) break; // key0가 released되면 반복문을 종료.
    }
}
```


임베디드시스템 설계 및 실습 - 보고서

```
        cnt++; // 반복문이 수행되는 횟수를 count.
        *ledr = cnt; // 반복문이 수행된 횟수를 ledr에 display.
        usleep(1000*1000); // 1초동안 대기 → 대략 1초에 한번씩 반복문이 수행되도록 한다.
    }
    if(cnt>9) cnt=9; // input integer의 overflow가 연산에 영향을 미치지 않도록
                    // overflow시 9의 값을 가지도록 한다.
    return cnt; // 반복문이 수행된 횟수를 반환한다.
                // (→ 대략 key0가 pressed된 시간(초)만큼의 value를 반환하게 된다.)
}

//// 수행될 연산을 하나의 key를 통해 입력 받게 된다.
int wait_for_operation(volatile int* key, volatile int* ledr, ₩
    int input1, int input2){
    int cnt=0;
    int tmp=0;
    while(1){ // key0가 released되는 순간을 polling.
        if(((key)&0x1)==0) break; // key0가 released되면 반복문을 종료.

        cnt++; // 반복문이 수행되는 횟수를 count.
        if(cnt==5) cnt=1; // cnt가 1~4의 값을 갖도록 한다.
        *ledr=cnt; // 반복문이 수행된 횟수를 ledr에 display.
        usleep(1000*1000); // 1초동안 대기 → 대략 1초에 한번씩 반복문이 수행되도록 한다.
    }

    //// cnt의 값에 따라 알맞은 operation 함수를 call한다.
    if (cnt==1) tmp = plus(input1, input2);
    else if (cnt==2) tmp = minus(input1, input2);
    else if (cnt==3) tmp = mul(input1, input2);
    else if (cnt==4) tmp = div(input1, input2);
    return tmp;
}

int main(void){
    int fd;
    void* lw_virtual;
    volatile int* key;
    volatile int* hex_3_0;
    volatile int* hex_5_4;
    volatile int* ledr;

    fd = open("/dev/mem", (O_RDWR | O_SYNC)); // memory device를 read, write 모드와 blocking IO 모드로 open.
    lw_virtual = mmap(NULL, LW_BRIDGE_SPAN, (PROT_READ | PROT_WRITE), ₩
        MAP_SHARED, fd, LW_BRIDGE_BASE); // lw_bridge의 base addr를 virtual address에 mapping.
    //// lw_virtual을 통해 각 peripheral address에 접근.

    key = (volatile int*) (lw_virtual + KEY_BASE);
    hex_3_0 = (volatile int*) (lw_virtual + HEX_BASE_3_0);
```

임베디드시스템 설계 및 실습 - 보고서

```
hex_5_4 = (volatile int*) (lw_virtual + HEX_BASE_5_4);
```

```
ledr = (volatile int*) (lw_virtual + LEDR_BASE);
```

```
int input1=0; // input1 value
```

```
int input2=0; // input2 value
```

```
int output=0; // output value
```

```
int operator_en=0; // operator enable signal (input과 operator를 순차적으로 입력 받게 된다.)
```

```
int input1_en=1; // input1 enable signal
```

```
int input2_en=0; // input2 enable signal
```

```
while(1){
```

```
    *hex_3_0 = segment(output) | segment(input2)<<16; // HEX display
```

```
    *hex_5_4 = segment(input1);
```

```
    if ((~operator_en) & (~input2_en) & (input1_en) & ((*key)!=0)){ // input1 입력.
```

```
        input1 = wait_for_value(key, ledr);
```

```
        input1_en = 0;
```

```
        input2_en = 1;
```

```
    }
```

```
    else if ((~operator_en) & (input2_en) & (~input1_en) & ((*key)!=0)){ // input2 입력.
```

```
        input2 = wait_for_value(key, ledr);
```

```
        input2_en = 0;
```

```
        operator_en = 1;
```

```
    }
```

```
    else if ((operator_en) & (~input2_en) & (~input1_en) & ((*key)!=0)){ // operator 입력(choose operation).
```

```
        output = wait_for_operation(key, ledr, input1, input2);
```

```
        operator_en = 0;
```

```
        input1_en = 1;
```

```
    }
```

```
}
```

```
munmap (lw_virtual, LW_BRIDGE_BASE);
```

```
// mapping된 virtual address 해제.
```

```
close(fd);
```

```
return 0;
```

```
}
```

```
#include <fcntl.h>
```

```
int open(const char* path_name, int flags, mode_t mode);
```

→ fd = open("/dev/mem", (O_RDWR | O_SYNC));

File을 정해진 mode로 열어 file descriptor를 생성하는 함수이다. linux에서의 file은 device, directory 등을 포함하는 확장된 의미

임베디드시스템 설계 및 실습 - 보고서

를 가지고 있다. O_RDWR은 read, write mode를 의미하고, O_SYNC는 해당 file에 값이 다 쓰여 질 때까지 다음 line을 진행하지 않겠다는 의미로, blocking IO mode를 의미하게 된다. 또한 file descriptor란 특정한 파일에 접근하기 위한 추상적인 키를 의미한다. 본 lab에서는 MMIO 형태로 program을 구현하였으므로 위의 두가지 mode로 memory device에 대한 file descriptor를 생성하여 사용하게 된다.

```
#include <sys/mman.h>

void *mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset);

int munmap(void* addr, size_t length);
```

→ lw_virtual = mmap(NULL, LW_BRIDGE_SPAN, (PROT_READ | PROT_WRITE), MAP_SHARED, fd, LW_BRIDGE_BASE);

<sys/mman.h>에 정의되어 있는 함수로, physical address를 virtual address에 mapping하는 역할을 하는 함수이다. Linux는 virtual memory system을 기반으로 설계되어 있기 때문에 device의 physical address를 바로 사용하여 access할 수 없다. 따라서 다음과 같은 mmap함수를 통해서 program에서 사용되는 virtual address에 lightweight bridge base의 physical address를 mapping하여 사용해야 한다(lab에서 사용되는 peripheral들은 모두 lw_bridge의 axi master를 통해 access 할 수 있으므로). 그 후에는 linux OS 내부에서 virtual address를 physical address로 바꾸어 사용하므로 올바르게 원하는 device에 access 할 수 있게 된다.

진행 결과

1.

```
root@delsoclinux:~/LAB3# gcc -c assign2.c -o assign2.o -marm
```

→ 위의 source code를 gcc를 통해 compile한다. 이 때 -marm option을 사용하여 cross compilation을 수행하게 된다.

2.

```
root@delsoclinux:~/LAB3# gcc assign2.o -o assign2
```

→ gcc를 통해 executable file을 생성한다.

3.

```
root@delsoclinux:~/LAB3# ./assign2
```

→ executable file을 실행한다.

임베디드시스템 설계 및 실습 - 보고서

4.



→ input1을 key0를 통해 입력하고 있는 모습이다. Key0를 누른 시간에 따라 ledr이 증가하게 되고 그에 따라 현재 입력하게 될 숫자를 확인할 수 있다. 사진에서는 '3'을 입력하고 있고, ledr에 나타난 값과 input1을 display하는 HEX4의 값이 동일한 것을 확인할 수 있다.

5.




→ input2를 key0를 통해 입력하고 있는 모습이다. Input1과 마찬가지로 ledr을 통해 값을 확인할 수 있고, input2를 display하는 HEX2와 ledr에 나타난 값이 동일한 것을 확인할 수 있다.





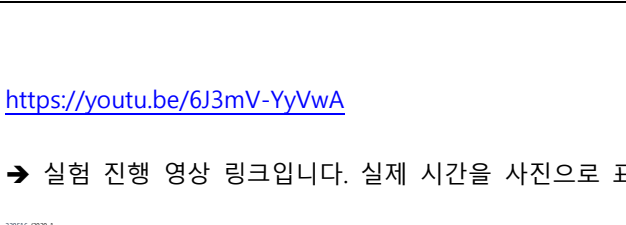
6. 사칙연산

plus		cnt==1일 때 수행되는 plus 연산이다. Input과 마찬가지로 시간에 따라 ledr이 증가되지만, 그 값이 1~4까지의 값을 modular하게 가지게 됨을 알 수 있다. Ledr에 나타난 값에 따라 연산의 종류가 결정되고, 그 결과가 HEX0에 display된다. HEX0에 display된 값을 통해 plus 연산이 제대로 수행되었음을 확인할 수 있다.
minus		cnt==2일 때 수행되는 minus 연산이다. HEX0에 display된 값을 통해 minus 연산이 제대로 수행되었음을 확인할 수 있다.
multiply		cnt==3일 때 수행되는 multiply 연산이다. HEX0에 display된 값을 통해 multiply 연산이 제대로 수행되었음을 확인할 수 있다.
divide		cnt==4일 때 수행되는 plus 연산이다. HEX0에 display된 값을

임베디드시스템 설계 및 실습 - 보고서

	<p>통해 divide 연산이 제대로 수행되었음을 확인할 수 있다.</p>
--	---

7. overflow

	<p>→ ledr에는 13이 display되지만, HEX4에는 9가 표현되는 것을 확인할 수 있다. 1 digit positive integer에 대한 연산이므로 input의 최댓값을 9로 설정하였다.</p>
<p>Input1 overflow</p>	
	<p>→ input overflow가 발생하면, 실제 입력인 13이 아니라 overflow 처리가 된 9에 대한 연산이 진행되는 것을 확인할 수 있다.</p>
<p>Input2 overflow</p>	
	<p>$7+6=13 \rightarrow 7+6=9$로 표현된 것을 확인할 수 있다. 1 digit positive integer에 대한 연산이므로 결과의 최댓값을 9로 설정하였다.</p>
<p>Output overflow</p>	
	<p>$2 * 7 = 14 \rightarrow 2 * 7 = 9$로 overflow된 결과가 최댓값인 9로 표현된 것을 확인할 수 있다.</p>
<p>Output overflow</p>	
	<p>$3 - 5 = -2 \rightarrow 3 - 5 = 0$으로 underflow된 결과가 표현된 것을 확인할 수 있다. 1 digit positive integer에 대한 연산이므로 결과의 최소값을 0으로 설정하였다.</p>
<p>Output underflow</p>	

<https://youtu.be/6J3mV-YyVwA>

→ 실험 진행 영상 링크입니다. 실제 시간을 사진으로 표현하기 부족하여 첨부합니다.

임베디드시스템 설계 및 실습 - 보고서

결과 분석 및 팀원 간 토의 사항

ARM architecture 기반의 DE1-SoC HPS platform에서 MMIO와 polled IO를 통한 device control을 하는 linux-program을 구현해 보았다. Linux는 virtual memroy system을 기반으로 설계가 되어있으므로, 일반적인 program에서 사용되는 address는 virtual address라고 할 수 있다. 그에 따라 access하고자 하는 physical address를 사용할 때 mmap이라는 함수를 통해서 해당 physical address를 virtual address에 mapping하는 과정이 필요했다.

또한 linux에서 file의 의미는 general file 뿐만 아니라 directory와 device를 포함하는 좀 더 범용적인 것을 의미하는데, 본 assignment에서는 MMIO를 통해 device를 control하기 위해서 open 함수를 통해 memory device에 대한 file descriptor를 생성하여 physical address를 mapping하는 데 사용하였다.

본 assignment에서 구현한 program은 key0를 polling하면서 입력을 받고 연산을 수행하며 HEX, ledr을 polling하며 결과를 display하는 program이다. Key0를 polling할 때 pressed event와 released event를 구분하여 polling하는 것이 이번 assignment에서 중요한 요소였다. Key0를 통해 어떠한 동작을 수행할 때는 pressed와 released event가 둘 다 발생하게 되므로 두 event를 적절히 감지하여 timing에 맞게 동작을 구현해야 했다. 또한 key0의 pressed time을 통해 입력을 받고, 연산을 수행하게 되는데 user가 인지하는 시간이 실제 시간과 달라 오류가 발생할 경우를 방지하기 위하여 ledr을 통해 user interface를 마련하는 것도 중요한 요소였다.

✓ 조원 별 기여 사항 및 느낀점

이름	기여도 (0 - 100%)	기여 사항	느낀점
박관영	33.3%	- lab2 코드 작성 및 보고서 작성	linux 기반의 program을 구현할 때 특정 함수를 사용하여 virtual address mapping을 하는 것을 경험할 수 있었고, MMIO를 통해 특정 peripheral에 access하여 control하는 것을 경험할 수 있었다. 또한 매번 device의 state를 확인하며 진행하는 primitive한 polled IO를 통해 device를 control하며 좀 더 cyclone V platform에 대한 직관적인 이해를 할 수 있었다.
오한별	33.3%	- lab2 코드 작성 및 보고서 작성	FPGA의 parallel port를 사용할 때 HPS와 FPGA 사이의 bridge를 통해 제어한다는 것을 배울 수 있었다. 또한 리눅스에서는 virtual memory를 사용하기 때문에 IO에 접근하기 위해서는 physical address로 변환하는 단계가 있어야 하는 것도 알게 되었다.
김서영	33.3%	- lab1 코드 작성 및 보고서 작성	HPS to FPGA Bridge를 통해 MMIO로 Peripheral을 제어하는 방법에 대해 학습할 수 있었다. Segmentation fault가 발생하였을 때 gdb를 이용하여 debubbing하는 방법에 대해 알 수 있었고 mmap함수의 사용법을 익힐 수 있었다.