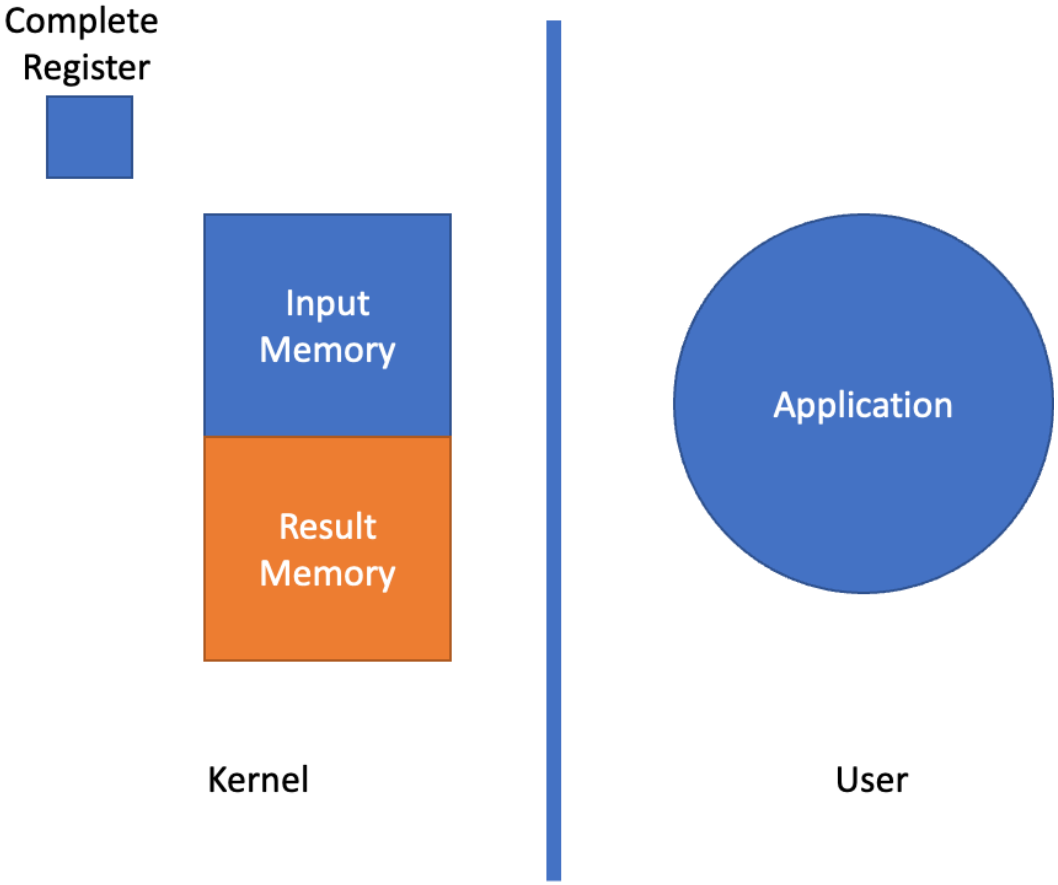
# 6주차 : Interrupt vs. Polling (2)

## 1. Test Scenario 구체화

저번 주와 달리 이번 주에는 테스트 시나리오를 좀 더 구체화 하였다. 테스트하고자 상황은 아래와 같다.

- NPU에서 계산이 되었다는 시그널을 Polling, interrupt를 사용하여 비교하고자 한다. NPU의 아주 대략적인 구조는 아래와 같다.




1. User Application쪽에서 Input Data를 넣는다.
2. User Application에서 start signal을 보낸다.
3. Kernel level에서 NPU 연산을 실시한다. 이때, NPU 연산은 1ms가 소요된다고 가정하자.
4. NPU 연산이 끝나면, Complete Register에 1을 쓴다.

**4-1 Polling : User에서 계속해서 register 값을 읽는다.**

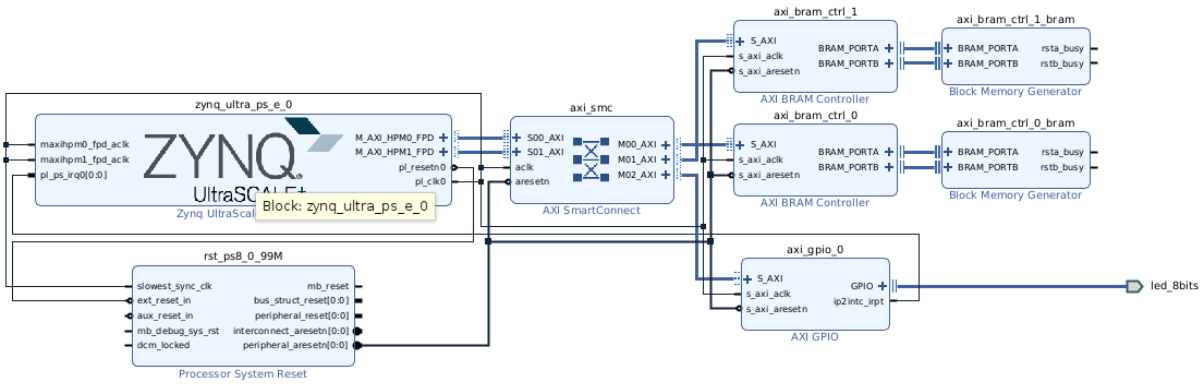
**4-2 Interrupt: register에 1이 쓰여지면 interrupt를 발생시킨다.**

5. 각 방식에 따라 Result를 User에게 제시한다.

여기서 측정하고자 하는 시간은 1번 소요시간, 2번 이후부터 5번 완료 까지의 시간을 측정해서 둘의 합산 시간을 측정하고자 한다. 산출 방식은 간단하게 10번 측정하고 평균치, 최대, 최소, 중간값을 결과로 낸다.

## 2. Polling

Polling의 경우 다음과 같이 구성하였다.




1. Input/Result을 담는 메모리를 BRAM(Block memory) IP를 사용하여, 이 memory를 mmap으로 mapping하여 사용한다. 메모리 복사는 memcpy()를 사용한다.
2. 사용자가 아무키나 누르면 그 시점부터 연산을 시작하는 것으로 간주한다. 아무 키나 누르면 thread가 분기하여 하나의 thread는 read를 일정 주기로 실시하여 polling을 수행한다. 한 쪽은 write를 보내 연산을 시작하고, kernel에서는 mdelay(1) 이후 LED를 on시킨다. 이는 계산이 완료되었다는 것을 의미한다.
3. Kernel에서는 계속해서 LED의 값을 user쪽으로 보내준다. Polling thread에서 1을 읽으면 thread를 종료시키고 원래의 thread에서 무한 루프를 탈출한다.
4. 이후 Result 값을 user에게 가져온다.

아래는 디바이스 드라이버 코드이다.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/unistd.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>
#include <linux/fs.h>
#include <linux/delay.h>
#include <linux/io.h>
#include <linux/device.h>
#define GPIO_BASE 504
#define DEVICE_NAME "LED"
#define MINOR_BASE 0
static dev_t my_dev;
static struct cdev my_cdev;
static struct class *my_class;
static int LED_open(struct inode *inode, struct file *file){
    printk(KERN_INFO "LED_open\n");
    return 0;
}
static int LED_release(struct inode *inode, struct file *file){
    printk(KERN_INFO "LED_release\n");
    return 0;
```

```
}
static ssize_t LED_write(struct file *file, const char __user *buf, size_t
count, loff_t *f_pos){
    printk(KERN_INFO "LED_write\n");
    if(gpio_get_value(GPIO_BASE) == 1)
        gpio_set_value(GPIO_BASE, 0);
    mdelay(1); // For calculation
    gpio_set_value(GPIO_BASE,1);
    return 0;
}
static ssize_t LED_read(struct file *file, char __user *buf, size_t count,
loff_t *f_pos){
    printk(KERN_INFO "LED_read\n");
    int value = gpio_get_value(GPIO_BASE);
    copy_to_user(buf, &value, sizeof(int));
    return 0;
}
static struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .open = LED_open,
    .release = LED_release,
    .write = LED_write,
    .read = LED_read,
};
static int LED_init(void){
    int ret;
    printk(KERN_INFO "LED_init\n");
    gpio_request(GPIO_BASE,"sysfs");
    gpio_direction_output(GPIO_BASE,0);
    gpio_export(GPIO_BASE,false);
    ret = alloc_chrdev_region(&my_dev, MINOR_BASE, 1, DEVICE_NAME);
    if(ret < 0){
        printk(KERN_ERR "alloc_chrdev_region failed\n");
        return ret;
    }
    cdev_init(&my_cdev, &my_fops);
    ret = cdev_add(&my_cdev, my_dev, 1);
    if(ret < 0){
        printk(KERN_ERR "cdev_add failed\n");
        return ret;
    }
    my_class = class_create(THIS_MODULE, DEVICE_NAME);
    if(IS_ERR(my_class)){
        printk(KERN_ERR "class_create failed\n");
        return PTR_ERR(my_class);
    }
    device_create(my_class, NULL, my_dev, NULL, DEVICE_NAME);
    return 0;
}
static void LED_exit(void){
    printk(KERN_INFO "LED_exit\n");
    device_destroy(my_class, my_dev);
    class_destroy(my_class);
    cdev_del(&my_cdev);
    unregister_chrdev_region(my_dev, 1);
    gpio_unexport(GPIO_BASE);
    gpio_free(GPIO_BASE);
```

```
}
module_init(LED_init);
module_exit(LED_exit);
MODULE_LICENSE("GPL");
```

아래는 User Application 코드이다.*

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <time.h>
#define BASE_ADDRESS 0xA0000000
#define BASE_ADDRESS_2 0xA0002000
#define LEN 8
static int condition = 0;
void* thread_func(void* arg){
    int res;
    do
    {
        sleep(1);
        read((int)arg,&res,sizeof(int));
    } while (res == 0);
    printf("Loop Escaped!\n");
    condition = 1;
}
int main(void){
    pthread_t thread;
    int tmp = 1;
    int thread_id;
    clock_t start1,end1,start2,end2;

    /*1st Step : Memcpy to BRAM*/
    int fd = open("/dev/mem", O_RDWR);
    int fd_2  = open("/dev/LED", O_RDWR);
    int fd_3 = open("/dev/mem",O_RDWR);
    char input;
    void* addr = mmap(0,LEN,PROT_READ | PROT_WRITE, MAP_SHARED, fd,
BASE_ADDRESS);
    void* addr_2 = mmap(0,LEN,PROT_READ | PROT_WRITE, MAP_SHARED, fd_2,
BASE_ADDRESS_2);
    if(addr == MAP_FAILED || addr_2 == MAP_FAILED){
        perror("mmap failed");
        close(fd);
        exit(0);
    }
    start1 = clock();
    memset(addr,0,LEN);
    end1 = clock();
    printf("1st step elapsed time is %ld\n",(end1-start1));
    /*2nd Step : Waiting User input...*/
    scanf("%c",&input);
    start2 = clock();
```

```
    /*3rd Step : Divide two thread*/
    thread_id = pthread_create(&thread, NULL, thread_func, (void*)fd_2);
    if(thread_id < 0){
        perror("Thread creation failed");
        munmap(addr,LEN);
        close(fd);
        exit(0);
    }
    write(fd_2,&tmp,sizeof(int));
    while(condition == 0)
    {
        printf("Waiting for polling completion...\n");
    }
    printf("Status is complete!\n");
    //memset(addr,0,LEN);
    memcpy(addr_2,addr,LEN);
    end2 = clock();
    printf("All process is completed!\n");
    printf("2nd step elapsed time is %ld\n",(end2-start2));
    munmap(addr,LEN);
    munmap(addr_2,LEN);
    close(fd);
    close(fd_2);
    close(fd_3);
    return 0;
}
```

위와 같은 코드를 작성하여 아래와 같은 결과를 얻어냈다.

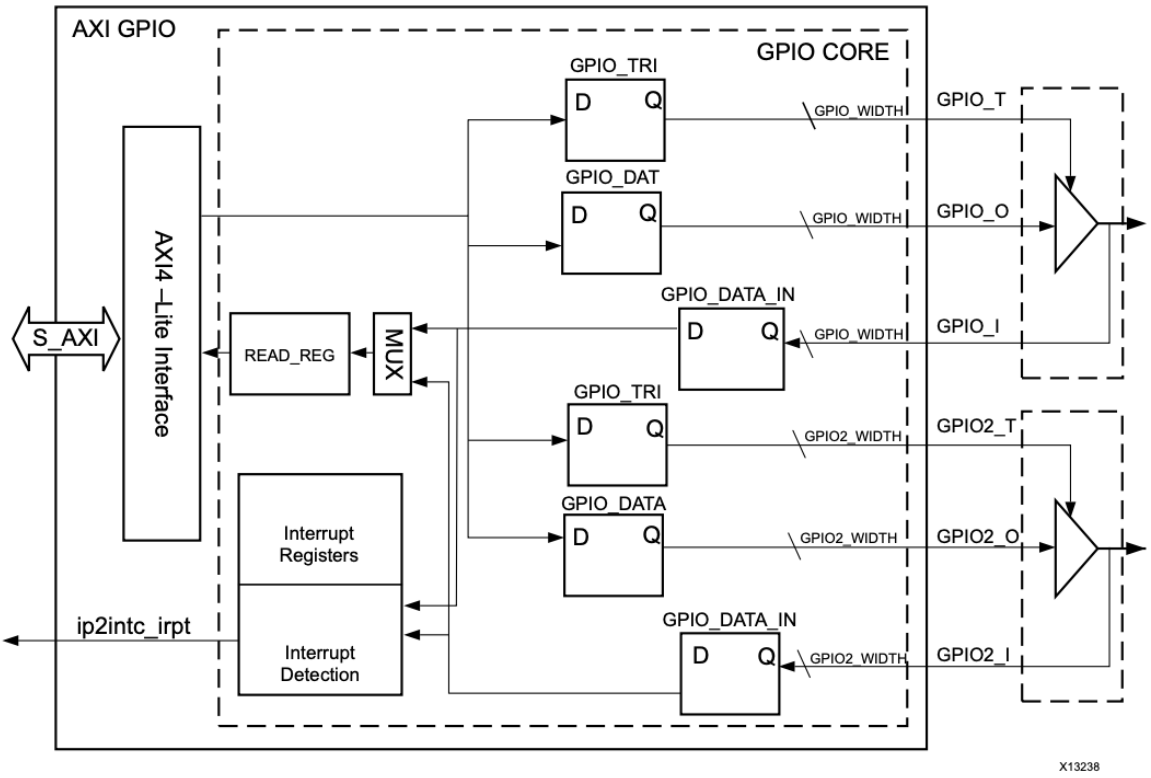
| | A phase (sec) | B phase (sec) | Total (sec) |
|---|---|---|---|
| **1** | 0.000007 | 1.006374 | 1.006381 |
| **2** | 0.000008 | 1.004457 | 1.004465 |
| **3** | 0.000018 | 0.972174 | 0.972192 |
| **4** | 0.00001 | 0.998068 | 0.998078 |
| **5** | 0.000008 | 0.992523 | 0.992531 |
| **6** | 0.000008 | 1.015169 | 1.015177 |
| **7** | 0.000008 | 1.008677 | 1.008685 |
| **8** | 0.000018 | 1.004888 | 1.004906 |
| **9** | 0.000009 | 1.013305 | 1.013314 |
| **10** | 0.000008 | 1.014039 | 1.014047 |
| **AVG** | 0.0000102 | 1.0029674 | 1.0029776 |
| **MAX** | 0.000018 | 1.015169 | 1.015177 |
| **MIN** | 0.000007 | 0.972174 | 0.972192 |
| **MEDIAN** | 0.000008 | 1.005631 | 1.0056435 |

단위가 초인것을 생각 했을 때 해당 수치는 굉장히 큰 수치라고 생각이 든다. 이 경우 고작 8KB만 데이터를 옮기고 복사하고 하였는데 1초면, 통상적으로 계산하는 양의 경우

에는 꽤 오래 걸린다고 생각이 든다. 감안해야 하는 점은, 일단 NPU 프로세서가 아니라는 것과 보드도 다른 것, 코드 최적화 요소도 감안해야 할 것이다. 즉, 이 값을 절대적인 수치의 평가보다는, 비교인 만큼 상대적인 수치로 비교하는 것이 합당하도 생각이 든다.

## 3. Interrupt (AXI GPIO)

초기에 design으로는 LED도 결국 value이므로 회로에서 wave로 생각하면 0에서 1로 올라가는 rising edge가 있을 것이다. 이를 감지하여 이것을 인터럽트로 쓰고자 하였고, 그 이유에서 polling에서도 AXI GPIO를 사용하였던 것이다. 실제로 인터럽트가 발생하는지 알아보기 위해 코드를 작성하여 확인해보았지만, 예상대로 결과가 나와주지 않았다. 결론적으로, AXI GPIO에서 물리적인 전기적 자극을 주지 않고 코드 상에서 value를 정해주는 것으로는 interrupt를 줄 수 없다고 결론짓게 되었다.




이 회로를 보고 그러한 결론을 내리게 되었는데, 위의 그림은 AXI GPIO의 Block Diagram이다. 우 하단부분의 Interrupt Detection 부분을 보면 interrupt를 감지 하는 block에 들어가는 input은 두개가 존재하며 두 개 모두 물리적 input, 즉 switch나 Push button의 전기적 신호가 필요한 것이다. 코드에서 value를 설정하는 것은 좌측의 AXI bus를 타고 들어오고, 이 값을 기반으로 Interrupt를 보내는 회로는 존재하지 않는다.
그렇다면 어떻게 계산이 다 되었다는 신호, 즉 여기서는 1ms를 측정할 것인가? 그래서 도입한 것이 AXI timer이다.

## 4. Interrupt (AXI Timer)

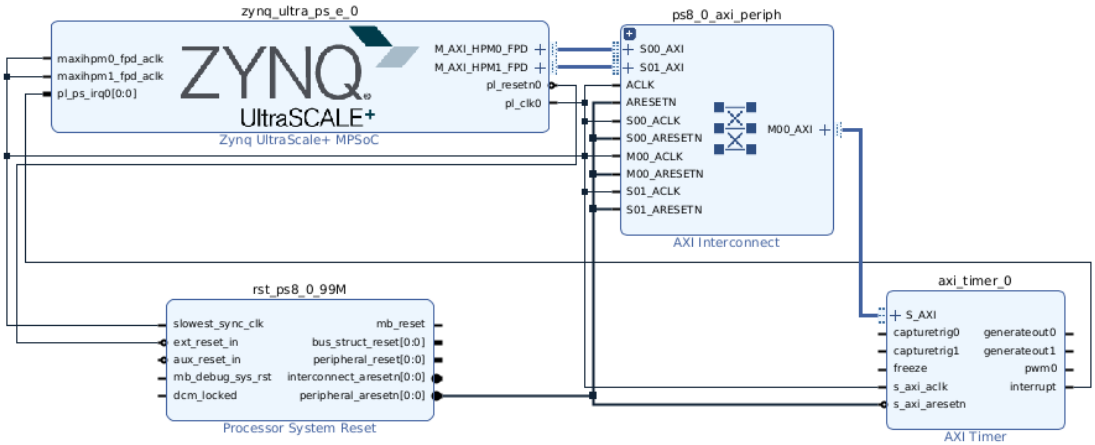
AXI Timer는 4가지 모드로 사용할 수 있다.

1. Generate mode

->흔히 아는 본래의 timer이다. 일정 값을 설정해두고 이 값을 증가 또는 감소 시켜 sign bit가 roll over되면 1을 generate한다. 이 모드를 사용할 것이다.

2. Capture mode
3. Pulse Width Modulation mode
4. Cascade mode

Generate mode를 사용하여 1을 generate 될 때마다 interrupt를 enable하면 같이 interrupt를 발생 시킬 수 있다.
그에 따라 다음과 같이 회로를 구성하였다.




Interrupt를 사용하기 위해서는 interrupt handler를 커널에 등록해야 한다. 등록하기 위해서는 Axi timer가 PS에 보내는 interrupt의 IRQ number을 알아야 등록 할 수 있다. 보통의 경우 PL -> PS interrupt는 121번부터 8개가 등록되며, 그 이후 나머지 8개가 등록되고, 16개의 interrupt를 사용할 수 있다. 이것이 petalinux에 반영 되면서 어떠한 interrupt냐에 따라 -16, -32까지 될 수 있으며, 이에 따라 AXI timer는 121 – 32 = 89가 배정받은 것으로 device tree에 나와있다. 하지만, 이 번호들을 사용하면 register_irq가 실패하는 것을 볼 수 있다. 저번 주 interrupt를 확인 할 때도, button의 interrupt 번호가 121, 89 둘 중 하나가 아닌 50번 이었던 것을 보아 무언가 사이에 변화가 있는 듯 하지만 이것을 찾아내지는 못하였다.

결국 가능한 interrupt 번호를 추리고, 그 번호들 중에서 timer인 interrupt를 찾아내는 방법밖에 존재하지 않았다. 그래서 아래와 같은 코드를 통해 번호들을 찾아내었다.

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/io.h>
#include <linux/device.h>
#include <linux/delay.h>
#include <linux/unistd.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>
#include <linux/fs.h>
#define DEVICE_NAME "timer"
#define MINOR_BASE 0
static dev_t my_dev;
static struct cdev my_cdev;
static struct class *my_class;
static unsigned int irqNum;
static irq_handler_t irq_handler(unsigned int irq, void *dev_id, struct pt_regs
*regs){
    printk(KERN_ALERT "Timer interrupted! \n");
    return (irq_handler_t) IRQ_HANDLED;
}
static ssize_t find_IRQ_write(struct file *filp, const char *buf, size_t count,
loff_t *f_pos){
    int iRQ;
    copy_from_user(&iRQ, buf, sizeof(int));
    int ret =
request_irq(iRQ,(irq_handler_t)irq_handler,IRQF_TRIGGER_RISING,"timer",NULL);
    if(ret < 0)
    {
        printk(KERN_INFO"%d is not proper IRQ # \n", iRQ);
    }
    else
    {
        printk(KERN_INFO"%d is proper IRQ # \n", iRQ);
    }
    return ret;
}
static struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .write = find_IRQ_write,
};
static int find_IRQ_init(void){
    int ret;
    ret = alloc_chrdev_region(&my_dev, MINOR_BASE, 1, DEVICE_NAME);
    if(ret < 0){
        printk(KERN_ERR "alloc_chrdev_region failed\n");
        return ret;
    }
    cdev_init(&my_cdev, &my_fops);
    ret = cdev_add(&my_cdev, my_dev, 1);
    if(ret < 0){
        printk(KERN_ERR "cdev_add failed\n");
        return ret;
    }
    my_class = class_create(THIS_MODULE, DEVICE_NAME);
```

```
    if(IS_ERR(my_class)){
        printk(KERN_ERR "class_create failed\n");
        return PTR_ERR(my_class);
    }
    device_create(my_class, NULL, my_dev, NULL, DEVICE_NAME);
    return 0;
}
static void find_IRQ_exit(void){
    device_destroy(my_class, my_dev);
    class_destroy(my_class);
    cdev_del(&my_cdev);
    unregister_chrdev_region(my_dev, 1);
    free_irq(irqNum,NULL);
}
module_init(find_IRQ_init);
module_exit(find_IRQ_exit);
MODULE_LICENSE("GPL");
```

아래는 User application 코드이다.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <time.h>
int main(void){
    int fd = open("/dev/timer",O_RDWR);
    if(fd < 0){
        perror("open failed");
        exit(0);
    }
    while(1){
        int res;
        int val;
        scanf("%d",&val);
        res = write(fd,&val,sizeof(int));
        if(res<0)
        {
            printf("NO.\n");
        }
        else
        {
            printf("OK. %d \n",val);
        }
    }
    close(fd);
    return 0;
}
```

이 코드는 사용자가 번호를 입력하면 이것을 커널로 전달하여 커널에서 핸들러 등록을 시도한다. Return 값에 따라 user에게 전해주는 값이 다르고 이 값을 기준으로 등록이 되는지 안되는지를 판별한다. 그 결과, 7,8,9,10,11,30,43,44,49,51,52가 등록되는 것을 확인하

였고, 이들 중에서 timer interrupt에 해당하는 번호는 49였다.
확인 방법은 아래와 같은 코드를 작성하여 확인하였다.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/io.h>
#include <linux/device.h>
#include <linux/delay.h>
#include <linux/unistd.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>
#include <linux/fs.h>
#define TIMER_ADDR_BASE 0xA0000000
#define TIMER_ADDR_HIGH 0xA000FFFF
#define TIMER_LOAD_REG_OFFSET 0x00000004
#define TIMER_VALUE 0x10000000
#define TIMER_MASK_AUTO_RELOAD 0x00000010
#define TIMER_MASK_LOAD_ENABLE 0x00000020
#define TIMER_MASK_INTR_ENABLE 0x00000040
#define TIMER_MASK_START 0x00000080
#define TIMER_MASK_CLEAR_INTR 0x00000100
#define DEVICE_NAME "timer"
#define MINOR_BASE 0
static unsigned int IRQNum;
static dev_t my_dev;
static struct cdev my_cdev;
static struct class *my_class;
static unsigned int irqnum = 49; // or 52? Finally, 49
unsigned int prev;
void* timer_addr;
static irq_handler_t irq_handler(unsigned int irq, void *dev_id, struct pt_regs
*regs){
    prev = ioread32(timer_addr);
    iowrite32(prev|TIMER_MASK_CLEAR_INTR,timer_addr);
    printk(KERN_ALERT "Timer interrupted! \n");
    return (irq_handler_t) IRQ_HANDLED;
}
static struct file_operations my_fops = {
    .owner = THIS_MODULE,
};
static int timer_init(void){
    /* Device driver Registration */
    int ret;
    ret = alloc_chrdev_region(&my_dev, MINOR_BASE, 1, DEVICE_NAME);
    if(ret < 0){
        printk(KERN_ERR "alloc_chrdev_region failed\n");
        return ret;
    }
    cdev_init(&my_cdev, &my_fops);
    ret = cdev_add(&my_cdev, my_dev, 1);
    if(ret < 0){
        printk(KERN_ERR "cdev_add failed\n");
        return ret;
    }
```

```
    my_class = class_create(THIS_MODULE, DEVICE_NAME);
    if(IS_ERR(my_class)){
        printk(KERN_ERR "class_create failed\n");
        return PTR_ERR(my_class);
    }
    device_create(my_class, NULL, my_dev, NULL, DEVICE_NAME);
    request_irq(irqnum, (irq_handler_t)irq_handler, IRQF_TRIGGER_RISING,
"timer_interrupt", NULL);
    /* Timer Setting */
    timer_addr  = ioremap(TIMER_ADDR_BASE,TIMER_ADDR_HIGH-TIMER_ADDR_BASE+1);
    if(timer_addr == NULL){
        printk(KERN_ERR "ioremap failed\n");
        return -1;
    }
    iowrite32(TIMER_VALUE,timer_addr+TIMER_LOAD_REG_OFFSET);
    /*Timer value setting*/
    iowrite32(TIMER_MASK_LOAD_ENABLE,timer_addr);
    /*Timer load enable, it should be clear.*/
    iowrite32(TIMER_MASK_INTR_ENABLE | TIMER_MASK_AUTO_RELOAD,timer_addr);
    prev = ioread32(timer_addr);
    iowrite32(prev|TIMER_MASK_START,timer_addr);
    /*Timer interrupt enable & auto reload, then timer start.*/
    return 0;
}
static void timer_exit(void){
    //iowrite32(0,timer_addr);
    free_irq(irqnum, NULL);
    iounmap(timer_addr);
    device_destroy(my_class, my_dev);
    class_destroy(my_class);
    cdev_del(&my_cdev);
    unregister_chrdev_region(my_dev, 1);
}
module_init(timer_init);
module_exit(timer_exit);
MODULE_LICENSE("GPL");
```

커널에서 AXI-Timer의 register를 조작하여 각종 값을 설정하고, 인터럽트 허용, 타이머 시작, 타이머 한 cycle 동작 후 auto reload등을 설정할 수 있다. 설정하는 방법은 timer 절대 주소를 ioremap으로 가상 주소를 받고, 이를 iowrite32, ioread32를 사용하여 bit masking을 이용하여 설정하였다.

그 결과 interrupt 번호를 49로 설정하였을 때 다음과 같은 결과를 얻어내었다.

```
[   78.828169] Timer interrupted!
[  119.097512] Timer interrupted!
root@xilinx-zcu106-2020_2:~#
```

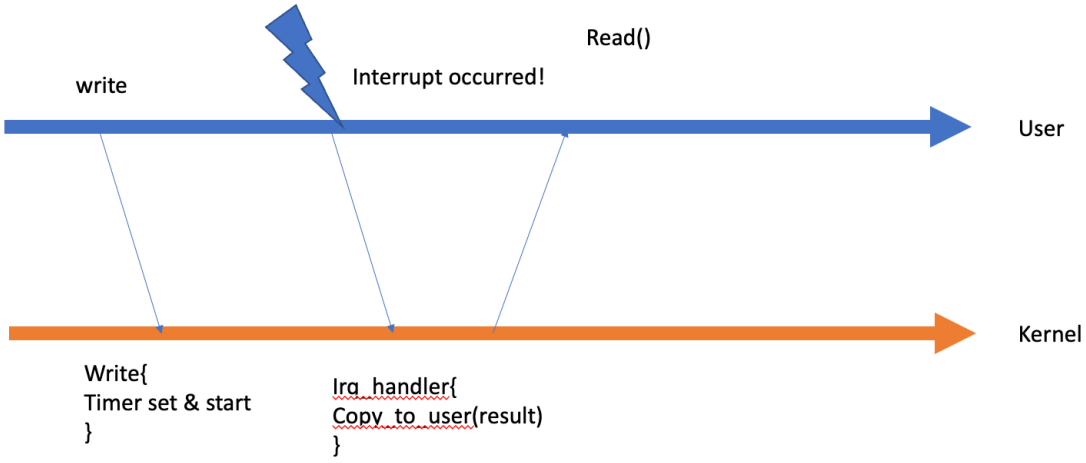
```
root@xilinx-zcu106-2020_2:~/irq# devmem 0xA0000000 w
0x000000D0
root@xilinx-zcu106-2020_2:~/irq# devmem 0xA0000008 w
0xA5E10BC7
root@xilinx-zcu106-2020_2:~/irq# devmem 0xA0000008 w
0xBC00AC57
root@xilinx-zcu106-2020_2:~/irq# devmem 0xA0000008 w
0xC27089AF
root@xilinx-zcu106-2020_2:~/irq# devmem 0xA0000008 w
0xC67222D9
root@xilinx-zcu106-2020_2:~/irq# devmem 0xA0000008 w
0xD1FC0CA6
root@xilinx-zcu106-2020_2:~/irq# devmem 0xA0000008 w
0xEA34ECE4
root@xilinx-zcu106-2020_2:~/irq# devmem 0xA0000008 w
0x108551CC
root@xilinx-zcu106-2020_2:~/irq#
```

0xA0000000은 word size로 timer의 control bit / status bit를 나타내며, 0xA0000008은 현재 카운터를 나타낸다. Devmem으로 확인 하였을 때 계속 같은 명령어를 입력하였을 때 값이 점점 증가하는 것을 볼 수 있고, MSB가 1에서 0으로 바뀌는 순간 interrupt가 발생하여 dmesg에서 interrupt message를 확인 할 수 있다. 다른 번호에서들은 interrupt handler가 작동하지 않아 그 내부에 있는 interrupt가 발생하였을 때 초기화 하는 과정이 실행되지 않아 다음 interrupt가 실행되지 않았다. 이는 devmem 0xA0000000을 찍었을 때, 1D0으로 계속 나타나는 것을 보고 확인 할 수 있는데, 인터럽트가 발생하면 저 1자리에 1을 다시 써주어 clear 해주어야만 다음 번 cycle에도 interrupt가 발생하기 때문에 이를 통해 확인 할 수 있다.

아래는 Axi timer control/status register의 8th bit에 대한 설명이다.

| | | | | |
|---|---|---|---|---|
| 8 | T0INT | Read/Write | 0 | Timer 0 Interrupt<br>Indicates that the condition for an interrupt on this timer has occurred. If the timer mode is capture and the timer is enabled, this bit indicates a capture has occurred. If the mode is generate, this bit indicates the counter has rolled over. Must be cleared by writing a 1.<br>*Read:*<br>0 = No interrupt has occurred<br>1 = Interrupt has occurred<br>*Write:*<br>0 = No change in state of T0INT<br>1 = Clear T0INT (clear to 0) |

인터럽트가 발생하는 것을 확인하였으나, 인터럽트 처리 이후 프로세스에 대해 의문이 생겼다. 다음과 같은 이유이다.




지금까지 구현 가능한 것은 irq_handler로 진입하는 코드까지이다. 그렇다면, 이 경우에서는 interrupt가 발생했다는 것은 계산이 끝난 것이므로 result를 user에게 전달해주어야 한다. Kernel Level에 존재하는 data를 user level로 전달하는 방법은 copy_to_user가 있는데, 이는 kernel과 user간의 통신 방법이므로 user에서 read를 호출해야 이 값을 받아 볼 수 있다. 하지만, 무슨 수로 user가 계산이 끝난 지점을 명확히 파악해서 그 시점에만 read를 호출할 수 있을 것인가? 결국 read를 호출하여 값을 받아보려면 while안에서 특정 조건에서만 Read를 호출하게 구현해야 하며 결국 loop가 존재하여 polling과의 대비점에서 이익을 볼 수 없게 될 것 같다는 생각을 하게 되었다.

이러한 상황에서 lock, conditional variable에 대한 조언을 듣게 되었다. 위 그림에서 user는 write를 부른 후 Blocking state로 들어가 cpu를 점유하고 있지 않다가, irq_handler에서 해당 프로세스를 scheduling하여 ready or run state로 끌어낸 다음 signal을 보내서 그 후 read를 호출하면 CPU 점유를 최소화 하면서 polling과의 차별 점도 살릴 수 있을 것 같다는 생각이 들었다. Context-switch라는 새로운 overhead가 존재하지만, 계속 cpu가 점유하면서 busy-wait을 도는 것보다는 minor한 요소라고 생각한다. 물론 아직 가설 단계라 좀 더 조사가 필요하며, 이를 다음주 목표로 정하고자 한다.

## 5. Discussion

- 비교의 올바른 방법

보통 어떠한 대상들에 대해 비교 실험을 할 경우 변인통제를 통해 대상의 차이점을 제외하고는 모든 부분을 공통되게 만들어 그 대상들의 특이 요소가 어떤 변화를 이끌어내는지를 파악하는 것이 중요하다. 이 경우에도 마찬가지라고 생각하였고, 변인을 계속해서 통제하려 하였으나, 무언가 계속 선입견이 들어가있는 상태로 구현을 하는 느낌이 들어 이 비교가 납득이 가능한 비교가 될까 라는 우려를 가지고 있다. 인터럽트를 구현하면서 무의식적으로 interrupt가 polling보다 좋아야 하는 것 아닌가 라는 생각이 들어 나중에 Interrupt를 다 구현한 후 review를 하여 변인이 통제 되었는지 확인이 필요할 것 같다.