Contents

3. Core Peripherals 3-1
3.1. 實驗目的 3-1
3.2. 實驗原理 3-1
3.2.1. About ARM Hardware Development Environment 3-1
3.2.2. Semihosting 3-4
3.2.3. Timer/Interrupt 3-6
3.3. 引導實驗 3-10
3.3.1. Scrutiny of Source Code: Interrupt Mechnism 3-10
3.3.2. Scrutiny of Source Code: Timer/Interrupt Memory Map 3-11
3.3.3. 實驗步驟 3-11
3.4. 實驗要求 3-15
3.5. 問題與討論 3-15
3.6. 參考文件及網頁 3-15

3. Core Peripherals

3.1. 實驗目的

This Lab let us be familiar with the ARM Hardware Development Environment, and try to understand the operation mechanism of semihosting and Timer/Interrupt. You will learn:
ARM Hardware Development Environment
Timer/Interrupt

3.2. 實驗原理

3.2.1. About ARM Hardware Development Environment

About ARM ASIC Platform (AP) Resources

An ATX motherboard which can be used to support the development of applications and hardware with ARM processor. A platform board provides the *AMBA* backbone and system infrastructure required. Core Modules & Logic Modules could be attached to ASIC Platform.

1. ARM Integrator/AP

   It includes system controller FPGA, clock generator, PCI bus interface supporting onboard expansion, External Bus Interface (EBI) supporting external memory expansion, Boot ROM, 32MB flash memory and 256K or 512K SSRAM。

   The system controller FPGA provides system bus to CMs and LMs, system bus arbiter, interrupt controller, peripheral I/O controller, 3 counter/timers, reset controller and system status and control registers

2. ARM Integrator/AP Architecture

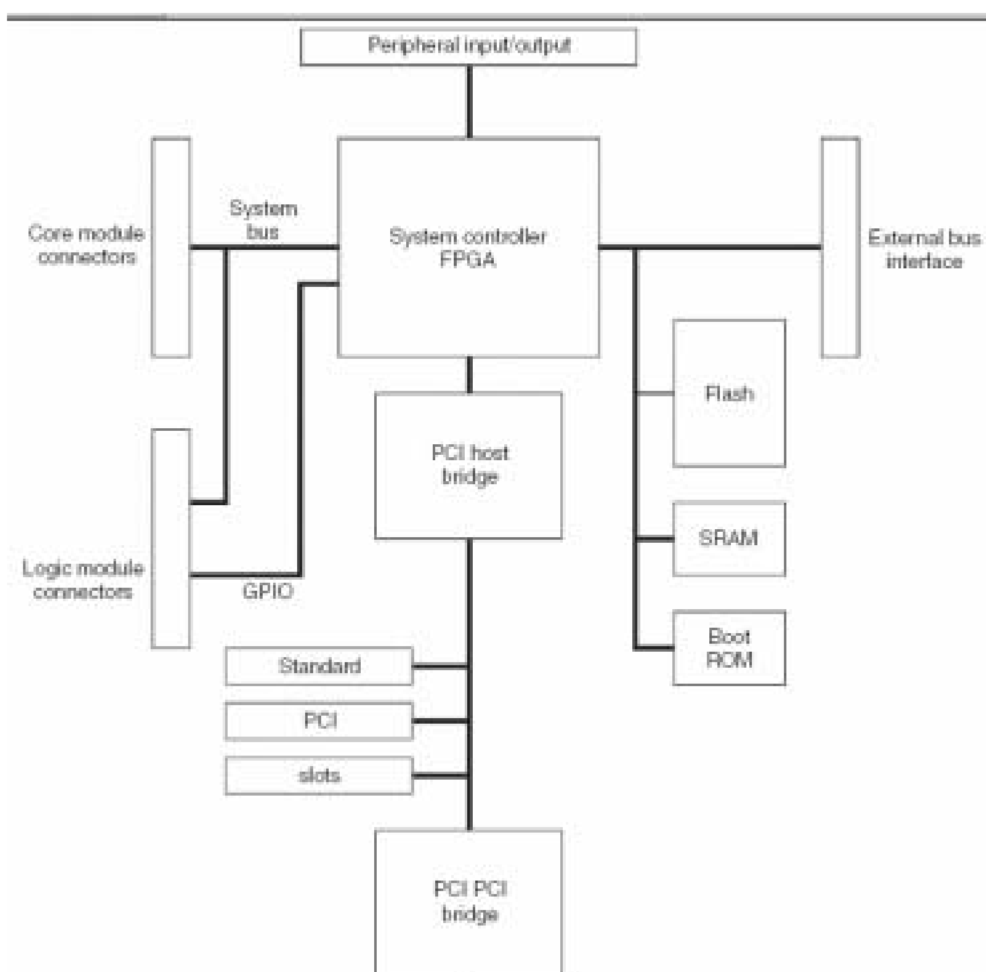

Figure 1 ARM Integrator/AP Architecture

2.1 ARM Integrator/AP System Controller FPGA

2.1.1 System Bus Interface:

It supports transfers between system bus and the Advanced Peripheral Bus (APB), transfers between system bus and the PCI bus and transfers between system bus and the External Bus Interface (EBI)

2.1.2 System Bus Arbiter.

Provides arbitration for a total of 6 bus masters. Up to 5 masters on CMs or LMs and PCI bus bridge can be the bus masters.

2.1.3 Peripheral I/O Controller

It includes 2 ARM PrimeCell UARTs, ARM PrimeCell Keyboard & Mouse Interface (KMI), ARM PrimeCell Real Time Clock (RTC), 3 16-bit counter/timers, GPIO controller and alphanumeric display, LED control and switch reader.

2.1.4 Reset Controller

Initializes the Integrator/AP when the system is reset

2.1.5 System Status and Control Register

Configure registers for clock speeds, software reset and Flash memory write protection)

2.1.6 Interrupt Controller FPGA

It handles IRQs and FIQs for up to 4 ARM processors. IRQs and FIQs originate form the peripheral controllers, OCI bus, and other devices on LMs. Assigns IRQs and FIQs from any sources to any of the 4 ARM processors. Interrupts are masked enabled, acknowledged, or cleared via registers in the interrupt controller. Main sources of interrupts includes system controller's internal peripherals, LM's devices, PCI subsystem and software.

2.1.7 System Controller FPGA Block Diagram

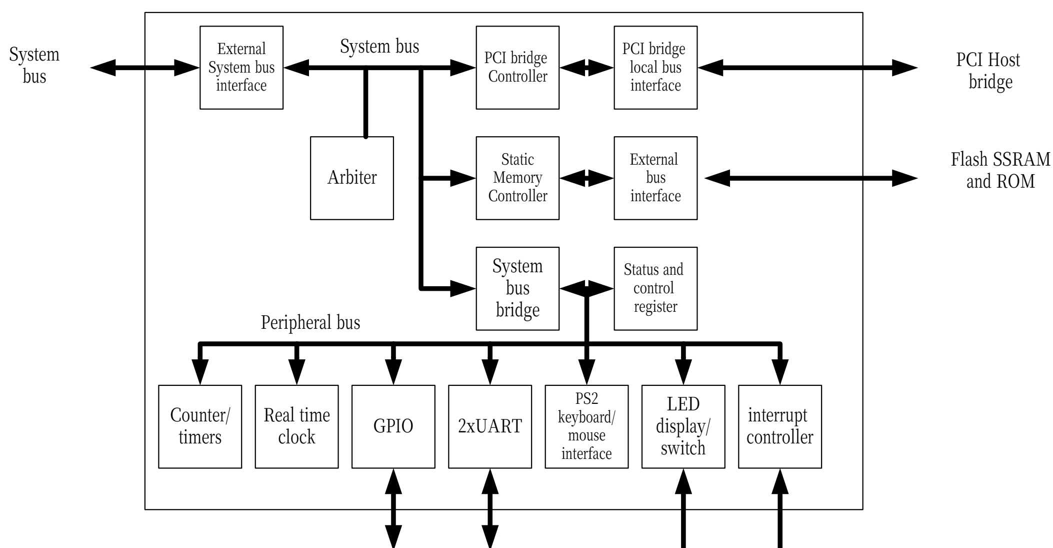

Figure 2 System Controller FPGA block diagram

ARM Integrator/Core Module (CM)

CM provides ARM core personality. CM could be used as a standalone development system without AP. Or CM could be mounted onto AP as a system core. CM could also be integrated into a 3rd-party development or ASIC prototyping system.

1. ARM Integrator/CM Features (CM7TDMI) ARM7TDMI microprocessor core.

It includes core module controller FPGA that performs the SDRAM controller, System bus bridge, Reset controller and Interrupt controller. It also supports 16MB~256MB pc66/pc100 168pin SDRAM , 256/512 KB SSRAM and Multi-ICE, logic analyzer, and optional trace connectors

2. ARM Integrator/CM Architecture

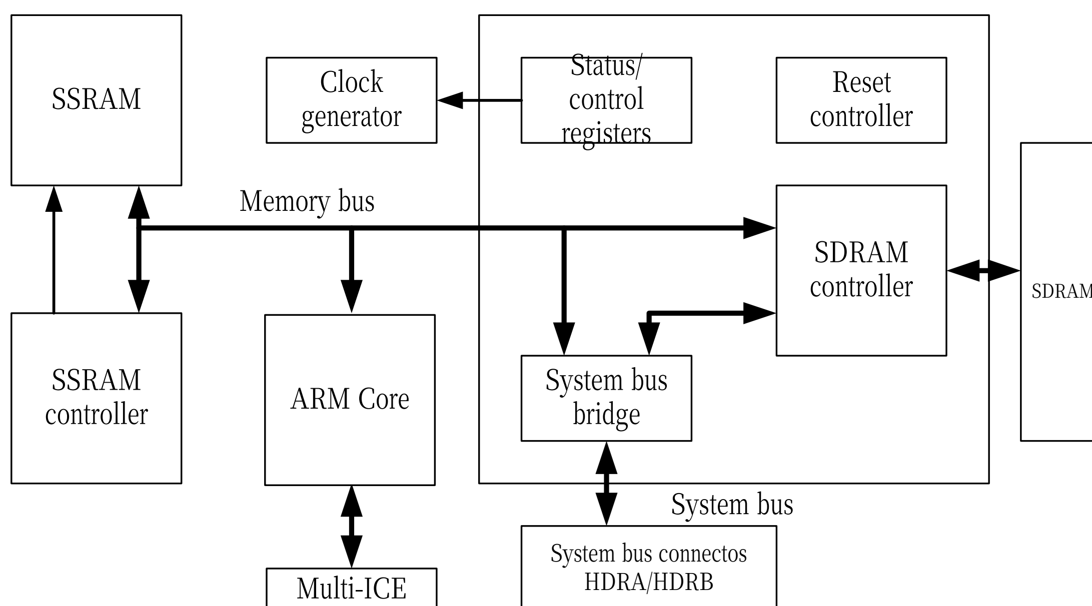

Figure 3 ARM Integrator/CM Architecture

3. Core Module FPGA
   It performs SDRAM controller: Supports for DIMMs from 16MB to 256MB. And it performs Reset controller: Initializes the core. Process resets from different sources. It has status and configuration space: Provides processor information. CM oscillator setup. Interrupt control for the processor debug communications channel. It also perform system bus bridge: Provides Interface between the memory bus on the CM and the system bus on the AP.

3.2.2. Semihosting

A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather than attempting to support the I/O itself.

When the developer attempt to show something through System I/O, the developer could let the application connected to a PC as a host with the debugger running. The debugger running on the host will handle the communications with the target application hardware, such as the ARM Integrator for example. The I/O request from the target application hardware will be handled and display by the host's debugger. This is called Semihosting.

Semihosting enables the developers to perform the system I/O through the host's debugger. The time and efforts for the developer to support system I/O request by writing hardware drivers is not required. This let the developer concentrate on the application development.

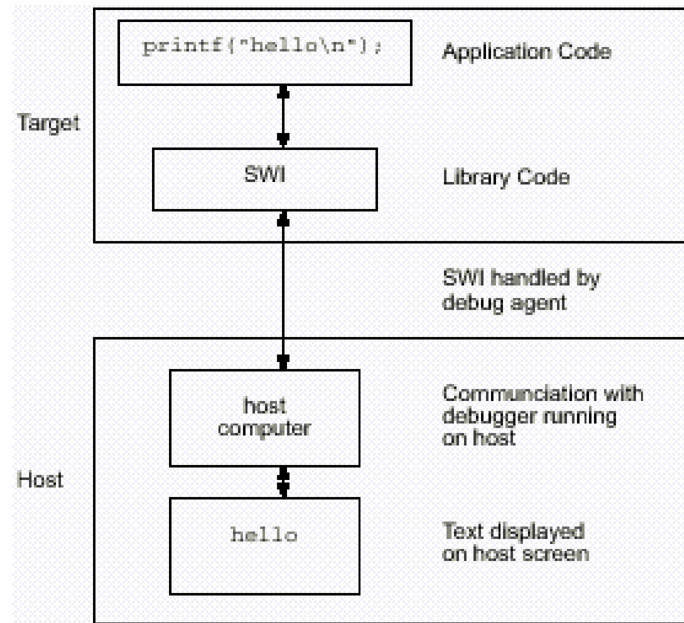

Figure 6. Semihosting overview.

1. How Semihosting Work?

   The application invokes the semihosting SWI(Software Interrupt). The debug agent then handles the SWI exception. The debug agent provides the necessary communication to the host system. Semihosting operations are requested using a semihosted SWI numbers:
   - 0x123456 in ARM state.
   - 0xAB in Thumb state.

2. Software Interrupt (SWI) Interface

   A Software Interrupt (SWI) is requested with an SWI number (Semihosting SWI numbers: 0x123456(ARM), 0xAB(Thumb)). Different operations in the SWI are identified using value of r0. Other parameters are passed in a block that is pointed by r1. The result is returned in r0. It could be an immediate value or a pointer.

3. Semihosting SWIs

   Semihosting operations used by C library functions such as printf(), scanf() uses semihosting SWIs. No need to implement semihosting operations for default standard I/O functions manually.

SWI	Description
SYS_OPEN (0x01) on page 5-12	Open a file on the host
SYS_CLOSE (0x02) on page 5-14	Close a file on the host
SYS_WRITEC (0x03) on page 5-14	Write a character to the console
SYS_WRITE0 (0x04) on page 5-14	Write a null-terminated string to the console
SYS_WRITE (0x05) on page 5-15	Write to a file on the host
SYS_READ (0x06) on page 5-16	Read the contents of a file into a buffer
SYS_READC (0x07) on page 5-17	Read a byte from the console
SYS_ISERROR (0x08) on page 5-17	Determine if a return code is an error

Figure 4 Semihosting overview.

3.2.3. Timer/Interrupt

This example installs a timer interrupt to update a variable. A loop in main() contains the code that reads the variable and outputs its value to the standard output port.
Observation key points are checking the Timer/Interrupt related registers values to see how they change, and observing how interrupt is handled.
The example must be run on the integrator to work. Using Armulator will not be able to show the correct results.

1. About Counter/Timers
   There are 3 counter/timers on an ARM Integrator AP. Each counter/timer generates an IRQ when it reaches 0. Each counter/timer has a 16-bit down counter with selectable prescale, a load register and a control register.

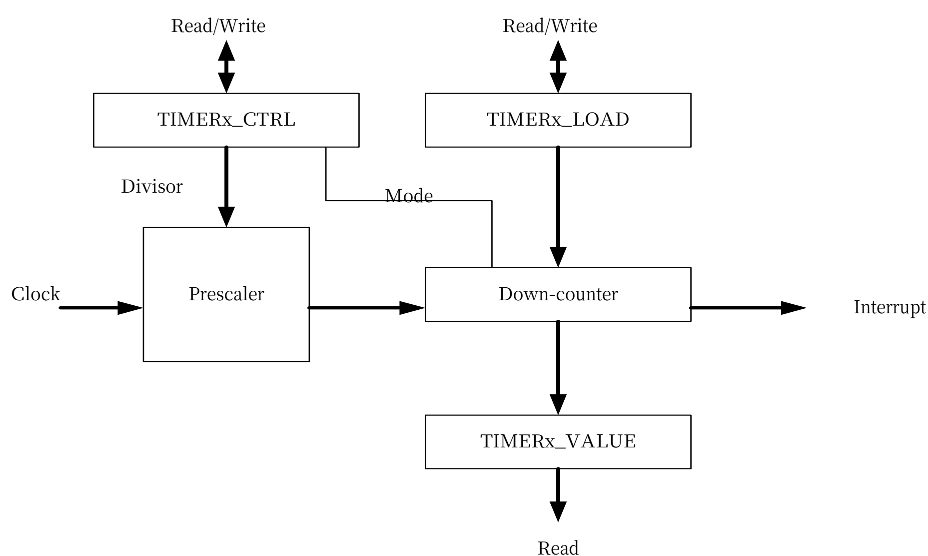

Figure 5 Timer/interrupt

2. Counter/Timer Registers

These registers control the 3 counter/timers on the Integrator AP board. Each timer has the following registers.

- TIMERX_LOAD: a 16-bit R/W register which is the initial value in free running mode, or reloads each time the counter value reaches 0 in periodic mode.
- TIMERX_VALUE: a 16-bit R register which contains the current value of the timer.
- TIMERX_CTRL: an 8-bit R/W register that controls the associated counter/timer operations.
- TIMERX_CLR: a write only location which clears the timer’s interrupt.

Address	Name	Type	Size	Function
0x13000000	TIMER0_LOAD	R/W	16	Timter0 load register
0x13000004	TIMER0_VALUE	R	16	Timer0 current value register
0x13000008	TIMER0_CTRL	R/W	8	Timer0 control register
0x1300000C	TIMER0_CLR	W	1	Timer0 clear register

Table 1 Timer registers description.

3. Timer Control Register

Bits	Name	Function
7	ENABLE	Timter0 load register
6	MODE	Timer0 current value register
5:4	Unused	Timer0 control register
3:2	PRESCALE	Prescale divisor: 00=none; 01=div by 16; 10=div by 256; 11=undefined
1:0	Unused	Unused, always 0

Table 2 Bits description of timer control register

4. About Interrupt Controller

Implemented in the system controller FPGA.Provides interrupt service routine dispatch for up to 4 processors(CMs).There's a 22-bit IRQ and FIQ controller for each processor. Each bit resembles an interrupt source.

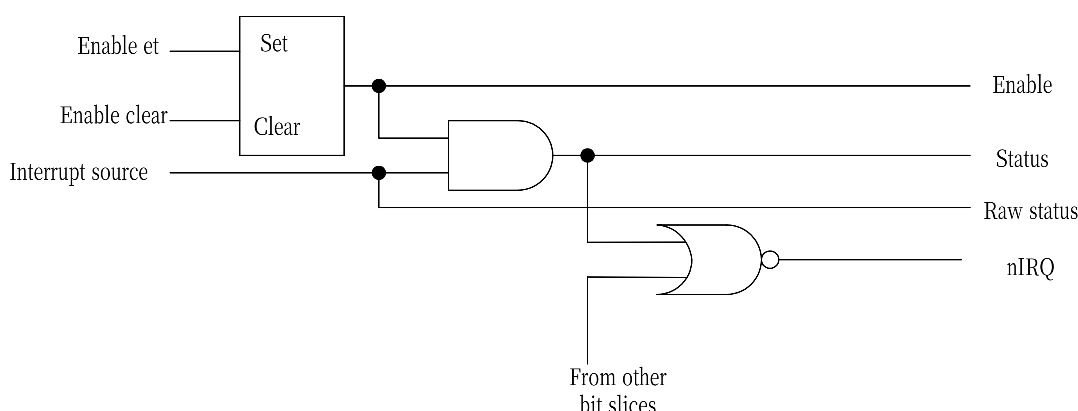

Figure 6 A bit slice of the interrupt control.

5. IRQ Registers

The registers control each processor's interrupt handler on the Integrator AP board.

Each IRQ has following registers:

- IRQX_STATUS: a 22-bit R register representing the current masked IRQ status.
- IRQX_RAWSTAT: a 22-bit R register representing the raw IRQ status.
- IRQX_ENABLESET: a 22-bit location used to set bits in the enable register.
- IRQX_ENABLECLR: a 22-bit location used to clear bits in the enable register.

Address	Name	Type	Size	Function
0x14000000	IRQ0_STATUS	R	22	IRQ0 status
0x14000004	IRQ_RAWSTAT	R	22	IRQ0 raw status
0x14000008	IRQ0_ENABLESET	R/W	22	IRQ0 enable set
0x1400000C	IRQ0_ENABLECLR	W	22	IRQ0 enable clear

Table 3 IRQ registers description.

6. IRQ Register bit assignments

Bits	Name	Function
7	TIMERINT2	Counter/Timer2 interrupt
6	TIMERINT1	Counter/Timer1 interrupt
5	TIMERINT0	Counter/Timer0 interrupt
4:1	Unused	Unused, always 0
0	SOFTINT	Software interrupt

Table 4 *Bits description of IRQ register.*

7. How Interrupt Works:

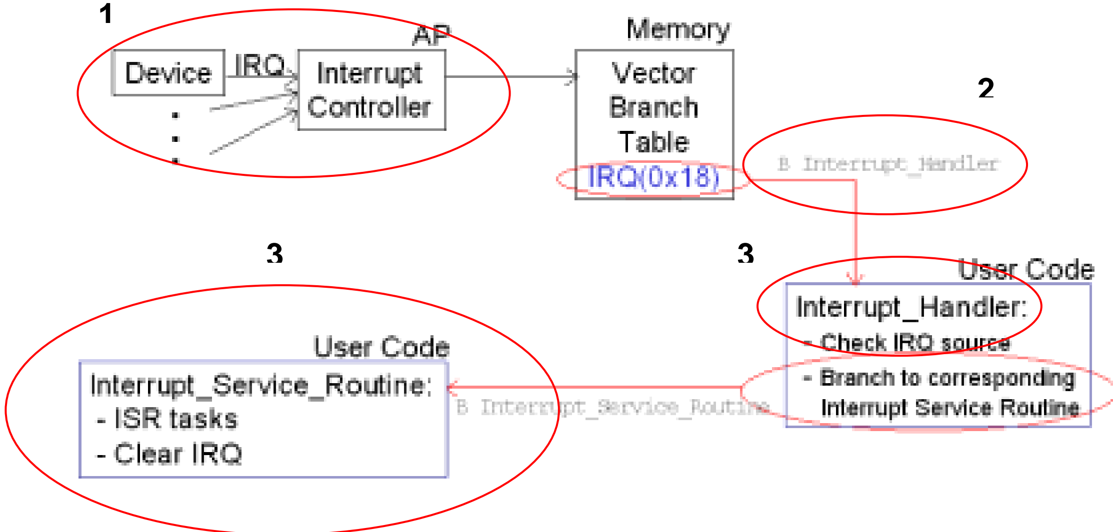

Figure 7 Interrupt processing flow

3.3. 引導實驗

After going through this leading experiment, you will understand the main mechanism of Timer/Interrupt operations.

There are two methods to perform the Timer/Interrupt operations: pure software or hardware method. .The software method can perform the operations purely on your desktop with the help of ARMulator and uHAL. You don’t have to port your codes onto the Integrator/AP but with the costs of some performance loss and functionality limit. The second method, hardware method, allows you to perform operations, including Timer/Interrupt, appropriately through semihosting and the aid of Multi-ICE. Either way is applicable. You may refer to the Labs for detailed information for software or hardware methods.

3.3.1. Scrutiny of Source Code: Interrupt Mechnism

In this lab, you have to figure out the operating principle and mechanism of Timer/Interrupt performed in the C program “irq.c.”

Important Functions:

- **Install_Handler**: This function installs the IRQ handler at the branch vector table at 0x18.
- **myIRQHandler**: This is the user’s IRQ handler. It performs the timer ISR in this example.
- **IRQ Mask Enable**: Set the IRQ0_ENA register to enable timer0 interrupt mask. So the IRQ could be accepted and handled.
- **enableIRQ**: The IRQ enable bit in the CPSR is set to enable IRQ.
- LoadTimer, WriteTimerCtrl, ReadTimer, ClearTimer: Timer related functions

To fully understand the hardware operations of ARM timers and their corresponding Interrupt operations, please be sure to (at least) survey the document DUI_0098B_AP_UG.pdf [1] of Section 3.5, 3.6 and 3.7.1. To be familiar with the software programming of Timer/Interrupt, please read Section 4.6 and 4.8 in the document.

At the beginning for the overview of “irq.c”, refer to the C code in circle 1. It sets the timer down-counting from 64 and triggers the timer to start down-counting. After a while, it down-counts to zero and triggers the interrupt signal IRQ to the Interrupt Controller in the AP (Circle 1 in Figure 7). IRQ corresponds to a branch vector of 0x18 and it branches the ARM processor to execute the Interrupt Handler (Circle 2 in Figure 7). But before execute

the Interrupt Handler, the programs has to install the handler as shown in circle 2 in the source code. You may refer to ADS_DeveloperGuide.pdf of Section 5.3 for more information. The Interrupt Handler of this program is the function __irq void myIRQHandler in circle 3 of the source code. It performs the Interrupt Service Routine (ISR) for the interrupt as shown in circle 3 of Figure 7. According to the source code (Circle 3), it just prints “HIHI!” and clear the timer’s IRQ. For more information, refer to ADS_DeveloperGuide.pdf of Section 5.5.

3.3.2. Scrutiny of Source Code: Timer/Interrupt Memory Map

To employ the timer and interrupt mechanism of the AP, your have to map the variables in the source code to specific memory addresses. Circle 4 in the source code installs the memory addresses of IRQ. Circle 5 enables the IRQs of Timer0 and SWI (0x21). The functions, LoadTimer, WriteTimerCtrl, ReadTimer, and ClearTimer install the memory addresses of corresponding registers and print out execution messages. Detailed information can be obtained in Section 4.6 and 4.8 in [1].

Note that if you implement this by the software method, you have to modify the mapped addresses according the specification in uHAL and generate you own make files. The procedure for the implement is shown in the OS Lab (Lab 4).

3.3.3. 實驗步驟

Create a new ARM Executable Image project, add irq.c to the project, make the project, and run the project finally.

```
#include <stdio.h>

unsigned Install_Handler( unsigned routine, unsigned *vector )
{
      unsigned vec, oldvec;
      vec = ((routine - (unsigned)vector - 0x8) >> 2 );
      /*->routine is the pointer point to the IRQ handler. */
      /*->shift right 2 is for address word aligned. */
      /*->subtract 8 is due to the pipeline */
      /*since PC will be fetching the 2nd instruction */
      /* after the instruction currently being executed. */
      vec = 0xea000000 | vec;
      /* to implement the instruction B <address> */
      /* 0xea is the Branch operation */
      oldvec = *vector;
      /* the IRQ address or FIQ address */
      *vector = vec;
/* the contents of IRQ address is now the branch instruction */
      return (oldvec);
 3

__irq void myIRQHandler (void)
{
      printf("\nFrom IRQ Handler>>HIHI!!\n");
      ClearTimer();     /* Clear the timer's IRQ */
}

// this function is used to set the I bit in CPSR
__inline void enable_IRQ(void)
{
  int tmp;
  __asm
  {
    MRS tmp, CPSR
    BIC tmp, tmp, #0x80
    MSR CPSR_c, tmp
  }
}
```

```
void d2b(int d_number, int array_len, int *b_number) {
 int len; /*array index*/      /* This function transform data
into binary digits */
 int temp=1;

 for (len=0;len<array_len;len++) {
     if (temp&d_number) b_number[len]=1;
     else b_number[len]=0;
     d_number=d_number>>1;
 }
}

void printB(int d_number, int array_len, int*b_number){
      int i; /* This function prints the binary digits */
      for(i=(array_len-1);i>=0;i--){
            printf("%d",b_number[i]);
            if ( i%8==0 && i!=0)
      printf("_");
      }
      printf("\n");
}

void LoadTimer(int loadvalue){
      int TIMER0_LOAD_ADDR = 0x13000000;
      int *TIMER0_LOAD;

      TIMER0_LOAD = (int *)TIMER0_LOAD_ADDR;

      *TIMER0_LOAD = loadvalue;
      printf("Timer Message>>> Timer0 loaded!!\n");
}

int ReadTimer(void){
      int TIMER0_VALUE_ADDR = 0x13000004;
      int *TIMER0_VALUE;
      TIMER0_VALUE = (int *)TIMER0_VALUE_ADDR;
      printf("Timer Message>>> Timer0 value aquired!!\n");
      return *TIMER0_VALUE;
}
```

```
void WriteTimerCtrl(int writevalue){
      int TIMER0_CTRL_ADDR = 0x13000008;
      int *TIMER0_CTRL;

      TIMER0_CTRL = (int *)TIMER0_CTRL_ADDR;

      *TIMER0_CTRL = writevalue;
      printf("Timer   Message>>>   Timer0   control   register
changed!!\n");
}

void ClearTimer(void){
      int TIMER0_CLEAR_ADDR = 0x1300000C;
      int *TIMER0_CLEAR;

      TIMER0_CLEAR = (int *)TIMER0_CLEAR_ADDR;

      *TIMER0_CLEAR = 1;
      printf("Timer Message>>> Timer0 cleared!!\n");
}

                  4
int main(void) {
      int IRQ0_STATUS_ADDR = 0x14000000;
      int IRQ0_RAWSTAT_ADDR = 0x14000004;
      int IRQ0_ENABLESET_ADDR = 0x14000008;
      int IRQ0_ENABLECLR_ADDR = 0x1400000C;

      int    *IRQ0_STATUS,    *IRQ0_RAWSTAT,    *IRQ0_ENABLESET,
*IRQ0_ENABLECLR;

      int b_num[22];
      int i;
   2  unsigned *irqvec = (unsigned *)0x18;

      Install_Handler( (unsigned)myIRQHandler, irqvec );
      /* Install user's IRQ Handler */

   4   enable_IRQ();          /* DR added - ENABLE IRQs */

      IRQ0_STATUS = (int *) IRQ0_STATUS_ADDR;
      IRQ0_RAWSTAT = (int *) IRQ0_RAWSTAT_ADDR;
      IRQ0_ENABLESET = (int *) IRQ0_ENABLESET_ADDR;
      IRQ0_ENABLECLR = (int *) IRQ0_ENABLECLR_ADDR;

   5
      *IRQ0_ENABLESET = 0x0021;
```

```
        d2b(*IRQ0_STATUS,22,b_num);
        printf("IRQ0_STATUS: ");
        printB(*IRQ0_STATUS,22,b_num);

        d2b(*IRQ0_RAWSTAT,22,b_num);
        printf("IRQ0_RAWSTAT: ");
        printB(*IRQ0_RAWSTAT,22,b_num);

        d2b(*IRQ0_ENABLESET,22,b_num);
        printf("IRQ0_ENABLESET: ");
        printB(*IRQ0_ENABLESET,22,b_num);

        d2b(*IRQ0_ENABLECLR,22,b_num);
        printf("IRQ0_ENABLECLR: ");
   1    printB(*IRQ0_ENABLECLR,22,b_num);

        LoadTimer(64);      /* set Timer0 reload value to 64 */
        WriteTimerCtrl(0xC4);     /* Enable the timer */
        // wait for a while
        for(i=0;i<1000000000;i++)
        {
                ;
        }
        printf("\nEND\n");
        return 0;
}
```

Figure 8 Source code of the lab.

3.4. 實驗要求

(1) Understand the mechanism of timer/interrupt. Use the timer/interrupt to evaluate the performance of other applications.
(2) Modified the C program. Try to use Real-Time Clock instead of Timer to show our IRQ0 values.

3.5. 問題與討論

How do you use multi-timer/interrupt?

3.6. 參考文件及網頁

1. Integrator ASIC Platform [DUI_0098B_AP_UG]
2. System Memory Map [DUI_0098B_AP_UG 4.1]
3. Counter/Timer [DUI_0098B_AP_UG 3.7, 4.6]
4. Interrupt [DUI_0098B_AP_UG 3.6, 4.8]
5. LEDs [DUI_0098B_AP_UG 4.5]
6. Core Module [DUI_0126B_CM7TDMI]
7. Core Module Registers[DUI_0126B_CM7TDMI 4.2]

8. Core Module Memory Organization [DUI_0126B_CM7TDMI 4.1]
9. SSRAM [DUI_0126B_CM7TDMI 3.2]
10. SDRAM [DUI_0126B_CM7TDMI 3.4]
11. SWI Interface [ADS_DebugTargetGuide 5.1.1]
12. SWI Handling [ADS_DeveloperGuide 5.4]
13. Semihosting [ADS_DebugTargetGuide 5]
14. Building Semihosted application [ADS_CompilerLinkerUtil 4.2]
15. Semihosting directly dependent functions [ADS_CompilerLinkerUtil Table4-1]
16. Semihosting indirectly dependent functions [ADS_CompilerLinkerUtil Table4-2]
17. I/O supported functions using semihosting SWI [ADS_CompilerLinkerUtil Table4-13]
18. uHAL API [AFS_Reference_Guide.pdf] [AFS_User_Guide.pdf]