CEG2400 - Microcomputer Systems

Lecture 8: More interrupts (revision)

Philip Leong

Introduction

- This lecture we will revisit the hello world program
- Main difference is that we will develop an interrupt driven version
- Entire program (linked with astartup.s) is available on course website
- I will first introduce a buggy version, show how to track down the bugs with the debugger and then give you the debugged version

Overview

- Initialization
  - Initialize LPC2131
- main()
  - Initialize UART0 and VIC
  - Infinite loop
- ISR
  - Sends next character

Initialize LPC21xx

- This is done in the startup code astartup.s
  - Exeception vectors
  - Reset handler
    - Sets up VPBDIV, PLL, stack for each mode, calls C code at __main
  - IRQ Handler

```
Vectors
    LDR   PC, Reset_Addr
    LDR   PC, Undef_Addr
    LDR   PC, SWI_Addr
    LDR   PC, PAbt_Addr
    LDR   PC, DAbt_Addr
    NOP   ; Reserved
    LDR   PC, [PC, #-0xFF0]
    LDR   PC, FIQ_Addr
```


main()

```
U0RBR    EQU    0xE000C000
U0THR    EQU    0xE000C000
U0IER    EQU    0xE000C004
U0IIR    EQU    0xE000C008
U0FCR    EQU    0xE000C008
U0LCR    EQU    0xE000C00C
U0LSR    EQU    0xE000C014
U0SCR    EQU    0xE000C01C
U0DLL    EQU    0xE000C000
U0DLM    EQU    0xE000C004
PINSEL0  EQU    0xE002C000
```

```
; VIC registers
VICIRQStatus EQU 0xFFFFF000
VICFIQStatus EQU 0xFFFFF004
VICRawIntr   EQU 0xFFFFF008
VICIntSelect EQU 0xFFFFF00C
VICIntEnable EQU 0xFFFFF010
VICIntEnClr  EQU 0xFFFFF014
VICSoftInt   EQU 0xFFFFF018
VICSoftIntClr EQU 0xFFFFF01C
VICProtection EQU 0xFFFFF020
VICVectAddr  EQU 0xFFFFF030
VICDefVectAddr EQU 0xFFFFF034
VICVectAddr0  EQU 0xFFFFF100
VICVectAddr1  EQU 0xFFFFF104
...
VICVectCntl0  EQU 0xFFFFF200
VICVectCntl1  EQU 0xFFFFF204
...
```


EQU

- This directive means “equate” or equals
- It is used to assign a numeric value in operand field to a name in the label field
- 0xE000C000 is a value in HEX
- You should try to use EQU’s rather than have “magic numbers” in your code
  - E.g. U0THR refers to uart0 transmitter holding register but if you write 0xE000C000, you don’t know what peripheral we are talking about
  - Also, if you get it wrong, it may be in many places

main()

```
; main segment
        ; tells assembler the following is code and not writeable
        AREA    main, CODE, READONLY
        EXPORT  __main
__main bl      init
loop    b       loop
```


AREA

- The ARM address space is divided into different areas e.g. RAM & FLASH
- We can put programs in flash, but not variables which should go in RAM
- The AREA directive tells assembler which parts of memory to put each chunk of code
  - In this case, it will go to flash as it is READONLY

Export

- Makes routine visible outside of this module
  - Needed because it is called from astartup.s

Branches

- Branch and link
  - Jumps to target address and saves return address in link register (r14)
- Branch
  - Jumps to address (requires pipeline flush)

Simple Quiz

- What does this do?

```
mov     r1, #0x5
ldr     r0, =PINSEL0
str     r1, [r0]
```


Harder Quiz

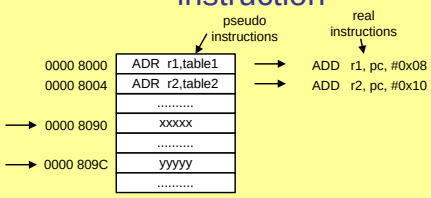
- What is the difference between

```
ldr     r0,=PINSEL0
ldr     r0,PINSEL0
```

Another Quiz

- How do I put 0x5 into r0?
- How do I put 0xE000C000 into r0?
- What is the difference between
  - ldr r0,=VAL
  - adr r0,VAL

Data Transfer Instructions - ADR instruction



- How does the **ADR** instruction work? Address is 32-bit, difficult to put a 32-bit address value in a register in the first place
- Solution: Program Counter PC (**r15**) is often close to the desired data address value
- **ADR r1, TABLE1** is translated into an instruction that add or subtract a constant to PC (**r15**), and put the results in r1
- This constant is known as **PC-relative offset**, and it is calculated as: addr_of_table1 - (PC_value + 8)

LDR rn,=val pseudo-instruction

- "LDR rn,=value" load arbitrary value into a register in most efficient way possible:
  - a simple value like 0xff assembled with a move instruction (like MOV rn,#0xff which takes 1 cycle on a standard ARM7)
  - a more complex instruction converted into a load from a program counter-relative address (such as LDR rn,[pc,#OFFSET] which takes 3 cycles on an ARM7)
- Seems complicated, but you don't need to worry about it: just use LDR rn,=CONST.
  - There is a little confusion created because LDR is not only a pseudo-instruction, but depending on the context an actual ARM assembler opcode mnemonic (LoaD Register).

Data Transfer Instructions Summary

- Size of data can be reduced to 8-bit byte with:

```
LDRB   r0, [r1]          ;  r0 : = mem8 [r1]
```

- Summary of addressing modes:

```
LDR    r0, [r1]          ; register-indirect addressing
LDR    r0, [r1 , # offset]   ; pre-indexed addressing
LDR    r0, [r1 , # offset]!  ; pre-indexed, auto-indexing
LDR    r0, [r1], # offset    ; post-indexed, auto-indexing
ADR    r0, address_label     ; PC relative addressing
```


uart

```
HSTR    =       "Hello world\n\r", 0
        align
;; initialises uart0 and vic
init
        mov     r1, #0x5        ; PINSEL0 = 0x5
        ldr     r0, =PINSEL0
        str     r1, [r0]

        mov     r1, #0x7        ; U0FCR = 0x7
        ldr     r0, =U0FCR
        strb    r1,[r0]

        mov     r1, #0x83       ; U0LCR = 0x83
        ldr     r0, =U0LCR
        strb    r1,[r0]

        mov     r1, #90         ; U0DLL = 15 (57600) or 90 (9600)
        ldr     r0, =U0DLL
        strb    r1,[r0]
```


uart

```
        mov  r1, #0x00      ; U0DLM = 0x0
        ldr     r0, =U0DLM
        strb    r1,[r0]

        mov  r1, #0x03      ; U0LCR = 0x03
        ldr     r0, =U0LCR
        strb    r1,[r0]

        ldr     r0,=HSTRp   ; init pointer for ISR
        ldr     r1,=HSTR    ; i.e. *HSTRp = HSTR;
        str     r1,[r0]
```

VIC

```
; now initialise VIC
        mov  r1,#0x0            ; VICIntSelect = 0x00 (all IRQ)
        ldr     r0,=VICIntSelect ; UART0 is channel 6
        str     r1,[r0]

        mov  r1,#0x40           ; VICIntEnable = 0x40
        ldr     r0,=VICIntEnable ; UART0 is channel 6
        str     r1,[r0]
```


VIC

```
        ldr     r1,=uart0_isr   ; VICVectAddr0 = &uart0_isr;
        ldr     r0,=VICVectAddr0 ; address of interrupt 0
        str     r1,[r0]

        mov  r1,#0x26       ; source 0 is UART0 i.e. #6
        ldr     r0,=VICVectCntl0 ; address of interrupt 0
        str     r1,[r0]
```


Quiz

- What needs to be programmed on the VIC in order to make it work?
- What is the difference between a vectored IRQ and a non-vectored IRQ in the VIC

Buggy Enable interrupts and start

```
mov  r1, #0x02  ; U0IER = 0x02 i.e. enable tx interrupts
ldr     r0, =U0IER
strb   r1,[r0]

bx      r14             ; return from subroutine
```


bx

- Return from a subroutine (similar to mov pc,r14 but works for thumb mode as well)

Buggy ISR

```
uart0_isr
        stmfd   sp!,{r0,r1,r2}
        ldr     r2,=HSTRp       ; address of pointer to next char
        ldr     r1,[r2]         ; value of HSTRp
        ldrb  r0,[r1]           ; is value zero?
        cmp  r0,#0
        ldreq r1,HSTR           ; if so restore HSTRp
        ldrbeq r0,[r1]          ; and get character
        add   r1,#1             ; advance to next position
        str     r1,[r2]         ; store in HSTRp
        ldr     r1,=U0THR       ; write next char, this also clears interrupt
        strb   r0,[r1]
        ldmfd sp!,{r0,r1,r2}
        subs  pc,lr,#4          ; return from interrupt
```

Quiz

- What exactly does the ARM processor do when it encounters an interrupt?
- What does the subs pc,lr,#4 exactly do?

Enable interrupts and start

```
mov     r1, #0x02  ; U0IER = 0x02 i.e. enable tx interrupts
ldr     r0, =U0IER
strb    r1,[r0]

mov     r0,#'I'         ; send first char
ldr     r1, =U0THR
strb    r0, [r1]

bx      r14             ; return from subroutine
```


Actual ISR

```
uart0_isr
        stmfd   sp!,{r0,r1,r2}
        ldr     r2,=HSTRp       ; address of pointer to next char
        ldr     r1,[r2]         ; value of HSTRp
        ldrb  r0,[r1]           ; is value zero?
        cmp  r0,#0
        ldreq r1,=HSTR          ; if so restore HSTRp
        ldrbeq r0,[r1]          ; and get character
        add   r1,#1             ; advance to next position
        str     r1,[r2]         ; store in HSTRp
        ldr     r1,=U0THR       ; write next char, this also clears interrupt
        strb   r0,[r1]
        mov  r0,#0xff           ; clear VIC interrupt
        ldr     r1,=VICVectAddr
        str     r0,[r1]
        ldmfd sp!,{r0,r1,r2}
        subs  pc,lr,#4          ; return from interrupt
```