Practice manual with Raspberry Pi2

PART II:

INTERRUPTS

Version 5.0

Prof. Julio Villalba Moreno
Department of Computer Architecture
University of Málaga
October 2021

3.- Interrups in the ARM processor of the Raspberry Pi2

An interrupt is a kind of exception in which an external asynchronous event breaks the normal execution of a program. This means that an interrupt can happen at any time and it forces the main program to branch to a new piece of code, in which a handler is found. Once the handler carries out its task, the control of the program is returned to the main program just to the place where it was interrupted. In other words, it is like a call to a routine externally enforced. To deal with interrupts, both the main program and the handler routine have to be prepared.

The basic tasks of the main programs are:

1. Initialize the vector table
2. Initialize the SP of the modes IRQ, FIQ and SVR
3. Configure peripherals, enable/disable local interrupts, enable global interrupts
4. Kernel of the main program (infinite loop)

The basic tasks of the handler are:

1. Push registers to be used
2. Perform handler work
3. Pop registers
4. Return to the main program

Let see all these steps to program an interrupt.

3.1.- Task of the main program

3.1.1.- Initialize the vector table

The ARM processor of the Raspberry Pi2 has 8 sources of exceptions. Each exception source forces the program counter (PC) to the address of the handler of that exception. In this address we have an interrupt vector which is an unconditional branch to the corresponding handler routine.

In this practice we are going to deal with two of those sources only: the IRQ (regular interrupt) and the FIQ (fast interrupt). The memory addresses for these interrupt sources are 18h and 1Ch respectively. Let ***IRQ_handler_routine*** denote the starting point of the handler routine of the interrupt IRQ, and ***FIQ_handler_routine*** denote the starting point of the handler routine of the interrupt FIQ. Thus, the memory addresses 18h and 1Ch have to content the corresponding unconditional branches:

Address	Content
0x00000018	**b IRQ_hadler_routine**
0x0000001C	**b FIQ_handler_routine**

To load these instructions on those positions we execute the macro instruction ADDEXC. To be exact, we have to write the following code for both interrupts:

```
        mov     r0, #0          @apunto tabla excepciones
        ADDEXC  0x18, irq_handler
        ADDEXC  0x1c, fiq_handler
```

3.1.2.- Initialize the SP of the SCV, IQR and FIQ modes

Once an interrupt is accepted, the processor changes the current operation mode (normally SVR) to that of the interrupt (either IRQ or FIQ in our system) automatically. In our practice we use three operation modes only: Supervisor mode (also call SVC, being the default mode of operation), IRQ mode and FIQ mode.

When the operation mode of the processor is changed due to an interrupt, the context of the program of the interrupted program (pc, sp, lr and cpsr) has to be saved because it has to be recovered after returning from the interrupt handler routine. To do that, each operation mode has three extra physical registers which keep some registers of the context of the interrupted program: LR_xxx, SP_xxx and CPSR_xxx to keep the lr, sp and cpsr of the main program (xxx means *irq* or *fiq*). Moreover, the FIQ mode has 5 extra general purpose registers denoted R8_fiq, R9_fiq, R10_fiq, R11_fiq and R12_fiq.

The processor has 37 physical registers. Only 18 registers of those registers (logical registers) are available at any time, depending on the selected operation mode. In this practice, when an operation mode is selected, 18 physical registers are selected and mapped on 18 logical registers according to the next table (logical registers are denoted in lowercase and physical registers in uppercase):

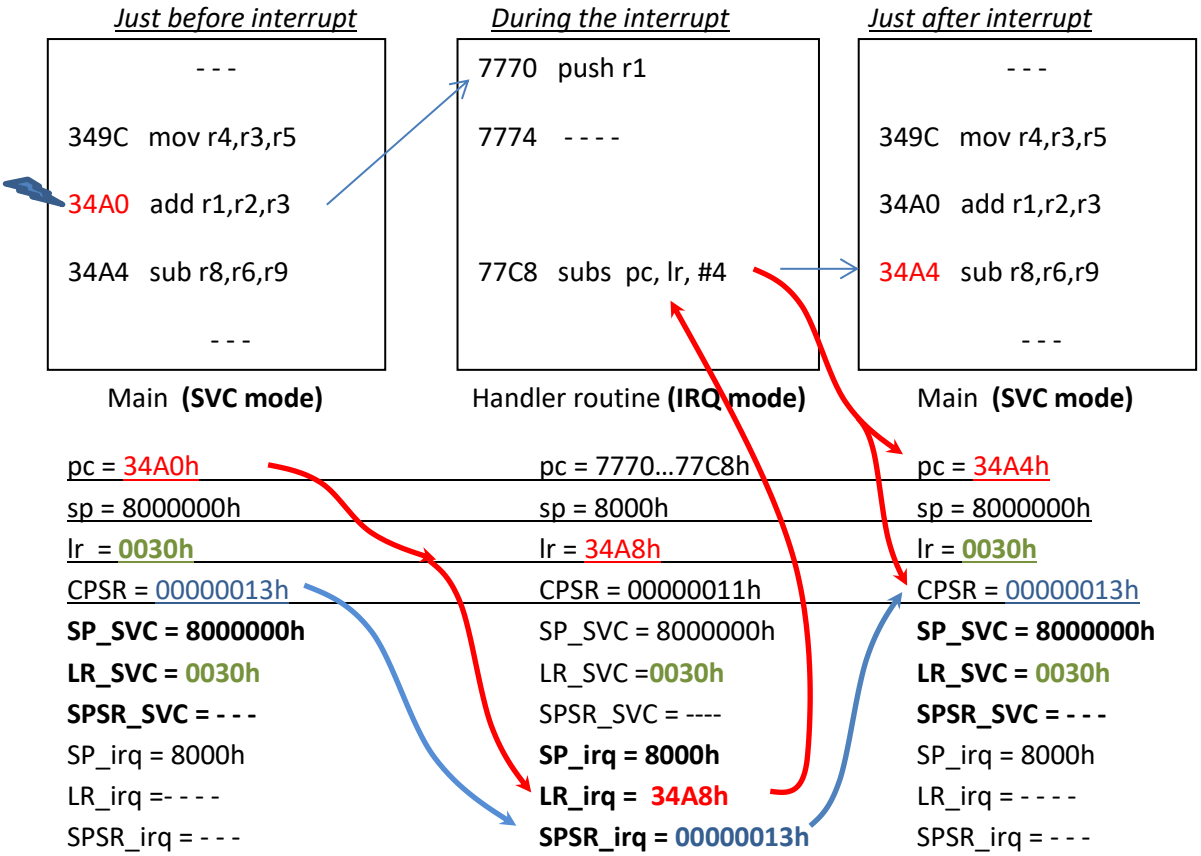
Logic register	SVC mode	IRQ mode	FIQ mode
	Physical register		
r0	R0	R0	R0
r1	R1	R1	R1
r2	R2	R2	R2
r3	R3	R3	R3
r4	R4	R4	R4
r5	R5	R5	R5
r6	R6	R6	R6
r7	R7	R7	R7
r8	R8	R8	$R8_{fiq}$
r9	R9	R9	$R9_{fiq}$
r10	R10	R10	$R10_{fiq}$
r11	R11	R11	$R11_{fiq}$
r12	R12	R12	$R12_{fiq}$
r13 (sp)	$R13_{svc}$ (SP_{svc})	$R13_{irq}$ (SP_{irq})	$R13_{fiq}$ (SP_{fiq})
r14 (lr)	$R14_{svc}$ (LR_{svc})	$R14_{irq}$ (LR_{fiq})	$R14_{fiq}$ (LR_{fiq})
r15 (pc)	PC	PC	PC
cpsr	CPSR	CPSR	CPSR
spsr	$SPSR_{svc}$	$SPSR_{irq}$	$SPSR_{fiq}$

Table I: Physical registers available for the three operation modes

When we are programing, we must be aware of the operating mode since different physical registers are used for the same logical registers. Summarizing, the 18 physical registers available for each mode are (see table above):

- SVC mode (supervisor mode, default mode for bare metal)
  - R0 – R12
  - SP_SVC: specific SP for the supervisor mode
  - LR_SVC: specific LR for the supervisor mode
  - PC
  - CPSR (Current Program Status register): Status register
  - SPSR_SVC (Saved Program Status register): keep a copy of the old status register (CPSR) of the main program
- IRQ mode:
  - R0 – R12
  - SP_irq: specific SP for the IRQ mode
  - LR_irq: keeps a copy of the old lr of the interrupted mode (LR_irq=LR_SVC)
  - PC
  - CPSR (Current Program Status register): Status register
  - SPSR_irq (Saved Program Status register): keep a copy of the old status register (CPSR) of the main program (SPSR_irq=CPSR_SVC)
- FIQ mode:
  - R0 – R7
  - R8_fiq - R12_fiq: 5 free general purpose registers available for the handler
  - SP_fiq: specific SP for the FIQ mode
  - LR_fiq: keeps a copy of the old lr of the interrupted mode (LR_fiq=LR_SVC)
  - PC
  - CPSR (Current Program Status register): Status register
  - SPSR_fiq (Saved Program Status register): keep a copy of the old status register (CPSR) of the main program (SPSR_fiq=CPSR_SVC).

For example, assume that an IRQ interrupt is produced when the main program was executing one instruction at the position 34A0h, with a set of registers as shown in below figure. Assume than the IRQ handler routine is placed in the address 7770h. Thus, just before the interrupt is processed the context is (pc, sp, lr, cpsr) = (34A0h, 8000000h, 0030h,13h). When the interrupt is accepted, the processor context changes to (pc, sp, lr, cpsr) = (7770h, 8000h, 34A8h, 11h). After returning from the handler, the context changes to (pc, sp, lr, cpsr) = (34A4h, 8000000h, 0030h,13h)

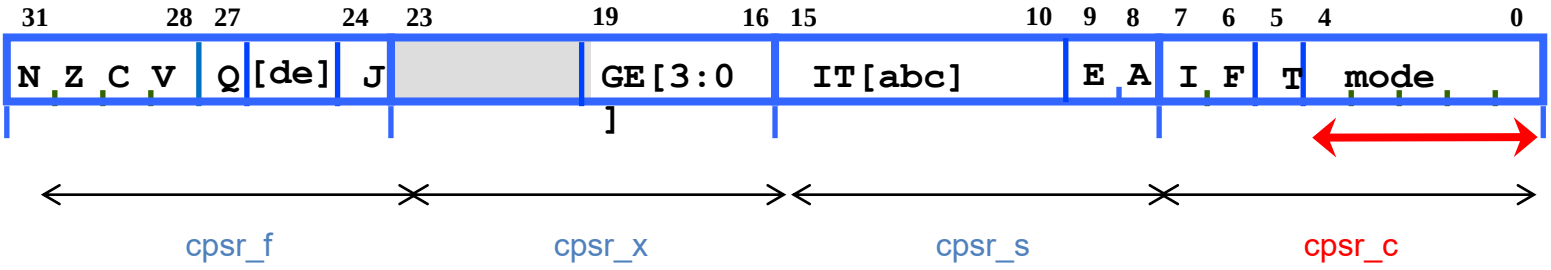


Note that the PC+8 is loaded in LR_irq instead of the actual return address (PC+4). This is due to the pipeline architecture of this ARM processor. To restore the actual return address the last instruction of the handler routine is ***subs pc, lr, #4.***

Another important issue here is that if we want make a call to a routine inside the interrupt handler routine (for example by using ***bl routine2***), it is necessary to save the current LR_xxx register in the stack by using the ***push*** instruction (and restore it by ***pop*** instruction):

```
Irq_handler:                    routine2:
        -----                           -----
        push{lr}                        -----
        bl routine2                     bx lr
        pop{lr}
        ----
```

Thus, before dealing with interrupts we have to initialize the SP of the three modes SVC, IRQ and FIQ modes. To do that, first we have to enter in the corresponding mode, and then enforce the SP. The operation mode is found in the 5 least significant bits (LSBs) of the status register (CPSR):



The codes of the operation modes are:

Code (bits 4-0 CPSR)	**Mode**
10000	User
10001	**Fast interrupt (FIQ)**
10010	**Regular interrupt (IRQ)**
10011	**Supervisor (SVC)**
10110	Sure Monitor
10111	Abort
11011	Undefined
11111	System

The CPSR is byte wise accessible, that is, we can modify individual bytes. In our case, we are interested in cpsr_c. Only two instructions can deal with the status register: ***msr*** and ***mrs***. For example, to enter in IRQ mode we have to enforce the 5 LSB to 10010 (see table above). Before entering in a mode, it is necessary to disable the IRQ and FIQ interrupt by forcing 1 the flag I and F of the CPSR (bits 7 & 6 of CPSR). Thus, we have to force cpsr_c = 11010010. Next sequence does that:

```
mov     r0, #0b11010010
msr     cpsr_c, r0
```

Once we are in IRQ mode, the SP register has to be loaded with the value 8 000 h:

```
mov     sp, #0x8000
```

Now, do the next exercises:

Exercise 1: Write down a code to initialize the SP of the FIQ mode to 4000h.

.

Exercise 2: Write down a code to initialize the SP of the SVC mode to 8 000 000h

Tasks of the handler

First of all, we have to store in the stack all the registers that we are going to use in the handler by the push instruction (1) and restore them by the pop instruction (5). Then, we have to find out the source of the interrupt (2) and do the specific task of the interrupt (3). Before coming back to the Main program, we have to clear the event (4) , restore the registers (pop (5)) and return from the interrupt by the instruction subs pc, lr, #4 (6). For example, if our handler uses the registers r0, r1 and r2:

irq_handler:

```
push {r0,r1,r2}
……
……
pop {r0, r1, r2}
subs pc, lr, #4
```

Nevertheless, for the fast interrupt (FIQ) there are a set of registers r8_fiq, r9_fiq, r10_fiq, r11_fiq and r12_fiq which are exclusive of this operation mode and they not need to be saved in the stack.

Structure of the code in detail

Our program has two parts: main program and handler routine. The main program has 8 points and the handler has 6 points, as shown below:

Main program (.text):

❶ Initialize Vector Table (IRQ/FIQ)
❷ Init the stack/s for FIQ/IRQ modes
❸ Init the stack for SVC mode (SVC mode selected)
❹ Configure GPIOs (I&O)
❺ Configure peripheral interruption: timer/push-buttons)
❻ Local enabling of configured interrupts
❼ Global enabling of interrupts (SVC mode)
❽ Infinite loop (polling of device/s?)

IRQ/FIQ Handler:

❶ Push registers to be used
❷ Source of interruption?
❸ Perform handler work depending on ❷
❹ Clear event (notify to device IRQ/FIQ has been served)
❺ Pop registers
❻ Return from handler

```
/* Basic skeleton for programs using interrupts */

.include "configuration.inc"
.include "symbolic.inc"

/* Vector Table inicialization */
     mov r0,#0
     ADDEXC 0x18, regular_interrupt @only if used
     ADDEXC 0x1C, fast_interrupt    @only if used


/* Stack init for IRQ mode */
     mov     r0, #0b11010010
     msr     cpsr_c, r0
     mov     sp, #0x8000
/* Stack init for FIQ mode */
     mov     r0, #0b11010001
     msr     cpsr_c, r0
     mov     sp, #0x4000
          mov r8,#0
/* Stack init for SVC mode */
     mov     r0, #0b11010011
     msr     cpsr_c, r0
     mov     sp, #0x8000000

/* Continue my MAIN program here */



end:   b end

/* Regular interrupt (only if used) */
regular_interrupt:
@    push { list of registers}


@    pop { list of registers}
     subs  pc, lr, #4

/* Fast interrupt (only if used) */
fast_interrupt:
@    push { list of registers}


@    pop { list of registers}
     subs  pc, lr, #4
```

3.1.3.- Configure peripherals, enable/disable local interrupts, enable global interrupts

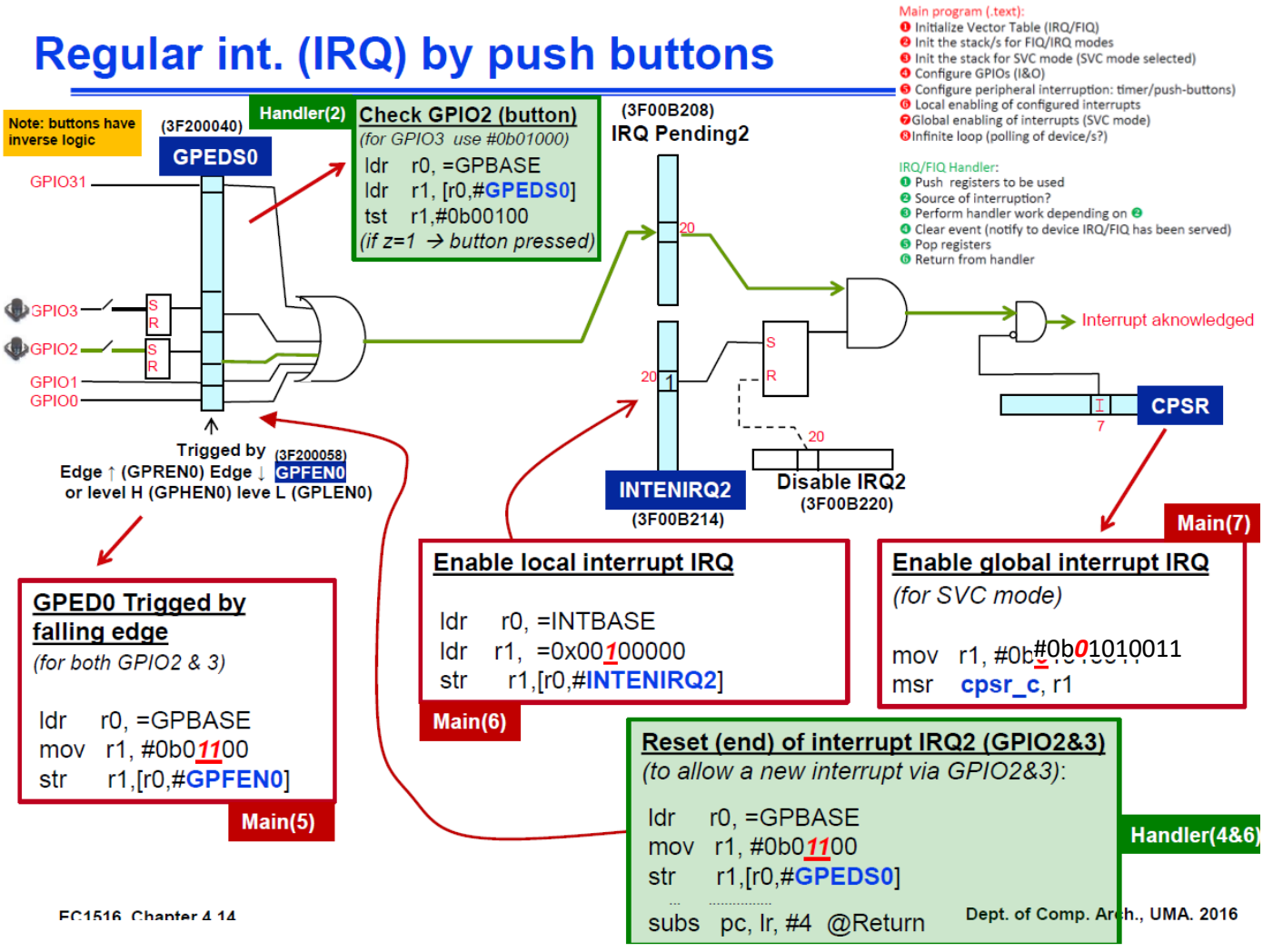
There are three sources of interrupts in our practice: push button 1, push button 2 and timer, and two kind of interrupts: regular (denoted IRQ) and fast (denoted FIQ).

3.1.3.1.- Configuring the hardware to allow interrupts by means of the push buttons

The push buttons are connected to the port 3F200040 (denoted GPEDS0) in the figure below. Once a push button is pressed, it is recorded in the bits 2 or 3 of the port GPEDS0, depending on the pressed button. Before doing this, the port GPEDS0 has to be configured as triggered by falling edge by setting the bits 2 & 3 of the port 3F200058 (denoted GPFEN0, see code in figure below).

In the figure below we can follow the sequence for regular interruptions **(IRQ)**. The OR gate drives the push buttons to the IRQ pending2 port (address 3F00B208, bit 20) which is connected to the upper input of the first AND gate. The lower input is connected to the bit 20 of the port 3F00B214 (denoted INTENIRQ2). Finally, the second AND gate corresponds to the global regular interrupt flag (I) which is the bit 7 of the status register CPSR.

Next figure also shows the sequence of assembly instructions to program a regular interrupt (IRQ) by the push buttons. Basically, follow the green line to get "interrupt acknowledge" and set 1 the inputs of the two AND gates by using ports.

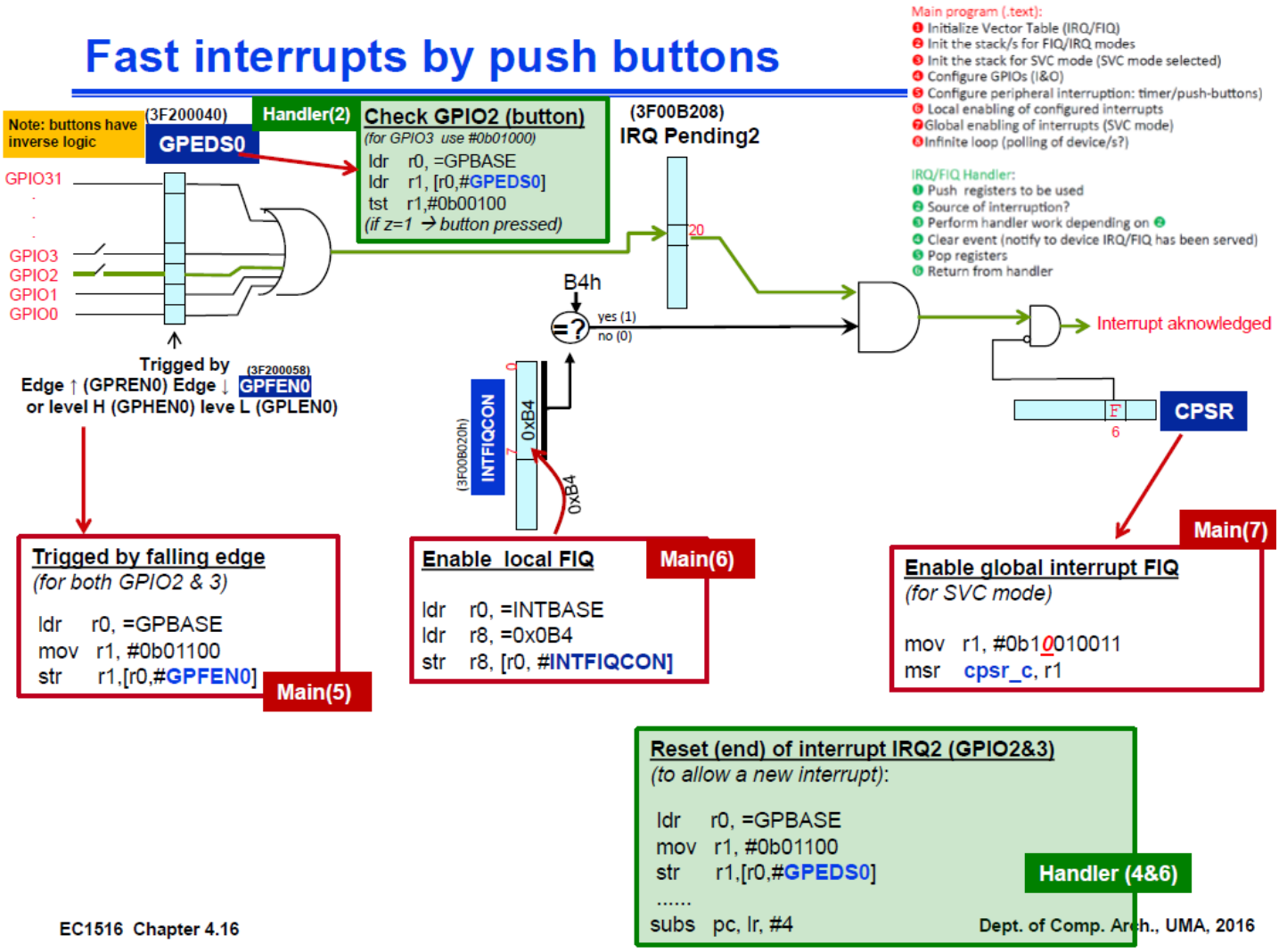


In the handler routine, first we have to check the pressed button (see the code in the figure). Finally, before leaving the handler routine, we have to reset the interrupt pending (see code above).

Exercise 3: Write a code which waits until one of the buttons are pressed to turn on a yellow led by IRQ

In the figure below we can follow the sequence for fast interruptions **(FIQ)**. In comparison with IRQ, now we have that the lower input to the first gate corresponds with the output of a 8-bit comparator (denoted by =? in the figure) such that if the code B4h is in the 8 LSB of port 3F00B020 (denoted INTFIQCON), then the output is 1. Finally, the second AND gate corresponds to the global fast interrupt flag (F) which is the bit 6 of the status register CPSR.

Next figure also shows the sequence of assembly instructions to program a fast interrupt (FIQ) by the push buttons. Basically, follow the green line to get "interrupt acknowledge" and set 1 the inputs of the two AND gates by using ports.



In the handler routine, first we have to check the pressed button (see the code in the figure). Finally, before leaving the handler routine, we have to reset the interrupt pending (see code above).

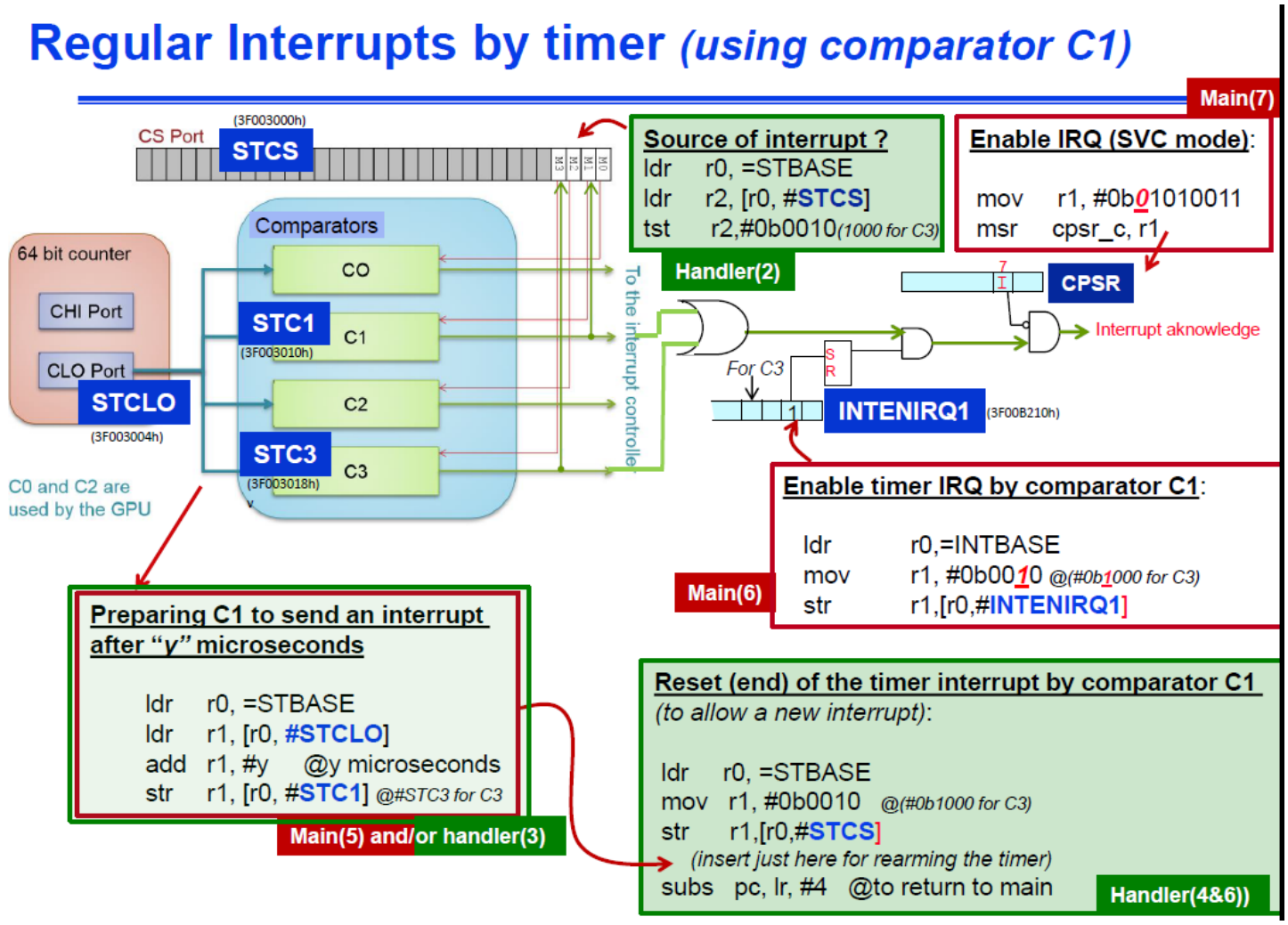
Exercise 4: Write a code which waits until one of the buttons are pressed to turn on a green red by FIQ

3.1.3.2.- Enabling the timer interrupts

For the timer, there are two sources available for us: comparator 1 (C1) and comparator 3 (C3) (comparators 0 & 2 are used by the GPU of the Raspberry). When the low part of the counter (port 3F003004h denoted STCLO) reaches the value of C1 (port 3F003010h denoted STC1), it sends an interrupt signal to the interrupt controller as well as notifies it by setting the bit 1 of the port 3F003000h (denoted STCS). Similarly, the comparator C3 is active when CL0 reaches C3 (port 3F003018h denoted STC3), and now the bit 3 of STCS is set.

In the figure below we can follow the sequence for regular interruptions **(IRQ)**. Once the interrupt is produced by the timer, there are two AND gates (masks) before an interrupt acknowledge is achieved (see figure below). The first one is connected to the bit 1 of the port 3F00B210h (denoted INTENIRQ1) and the second one corresponds to the global regular interrupt flag (I) which is the bit 7 of the status register CPSR.

Next figure also shows the sequence of assembly instructions to program the hardware for C1. Basically, follow the green line to get "interrupt acknowledge" and set 1 the inputs of the two AND gates by using ports.

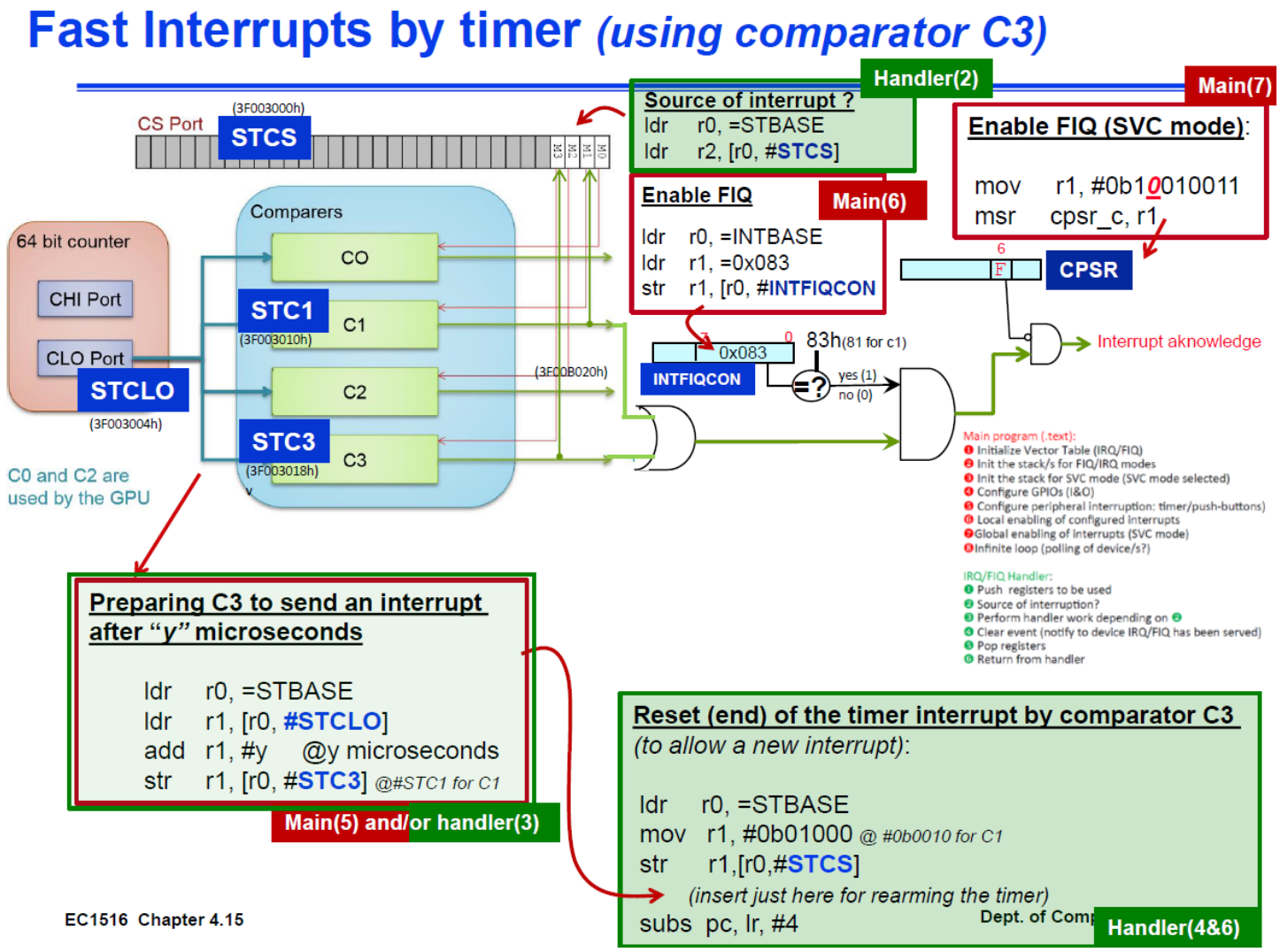


To program a delay of "y" microseconds, we have to read the current value of CLO, add "y" and store the result in C1 (see code in the figure above). To distinguish if a timer interrupt is produced by C1 or C3, we have to check the bits 1 and 3 of the port STCS in the handler routine (see it later). **When rearming the timer in the handler routine, it is necessary that you first reset the source of interrupt (STCS port) and then rearm the timer (access STC1 or STC3).**

Exercise 5: Write to generate a tone of 250 Hz using the timer and IRQ

In the figure below we can follow the sequence for fast interruptions **(FIQ)**. Once the interrupt is produced by the timer, there are two AND gates (masks) before an interrupt acknowledge is achieved. The first AND gate is connected to the output of a 8-bit comparator (denoted by =? in the figure) such that for comparator 3, if the code 83h is in the 8 LSB of port 3F00B020 (denoted INTFIQCON), then the output is 1 (for comparator 1 use code 81h); on the other hand, the other input is connected to the OR gate from the comparators C1 and C3. Finally, second AND corresponds to the global fast interrupt flag (F) which is the bit 6 of the status register CPSR.

Next figure also shows the sequence of assembly instructions to program the hardware for C3. Basically, follow the green line to get "interrupt acknowledge" and set 1 the inputs of the two AND gates by using ports.



To program a delay of "y" microseconds, we have to read the current value of CLO, add "y" and store the result in C3 (see code in the figure above).). In this case, the source of interrupt is unique (you program C3 or C1 only, but not both at the same time). **When rearming the timer in the handler routine, it is necessary that you first reset the source of interrupt (STCS port) and then ream the timer (access STC1 or STC3).**

Exercise 6: Write to generate a tone of 1000 Hz using the comparator 3 of the timer and FIQ

3.2.- Tasks of the handler

First of all, we have to store in the stack all the registers that we are going to use in the handler by the push instruction, and restore them by the pop instruction. For example, if our handler uses the registers r0, r1 and r2:

```
irq_handler:

        push {r0,r1,r2}

        ……

        ……

        pop {r0, r1, r2}

        subs pc, lr, #4
```

Nevertheless, for the fast interrupt (FIQ) there are a set of registers r8_fiq,r9_fiq,r10_fiq,r11_fiq and r12_fiq which are exclusive of this operation mode and they not need to be stored in the stack.

4.- Use of variables

You can define variables in the memory system (required when you need extra registers). For example, to declare VAR1 as a variable initialized to 0 we write

```
VAR1:   .word  0
```

To read this variable, you have to write this code (for example, to load r2 with the value of the variable VAR1):

```
ldr     r1,=VAR1
ldr     r2,[r1]
```

Now, the value of VAR1 is in the register r2. To write VAR1 with a constant value, we can write:

```
ldr     r2,#constant
ldr     r1,=VAR1
str     r2,[r1]
```

Now the constant is stored in VAR1.

- *Note: define your variable at the end of your program, out of the code zone*