Practice manual with Raspberry Pi2

PART I:

INPUT/OUTPUT PORTS

Version 5.0

Prof. Julio Villalba Moreno
Department of Computer Architecture
University of Málaga
November 2019

The purpose of this practice is to manage the input/output system of an ARM processor. To do that, we use a board which contents such processor: the Raspberry Pi2. This board can be configured to work with the assembly instructions of the ARM processor directly (bare metal mode). Furthermore, several external devices (6 LEDs, two push buttons and one speaker) have been connected to the Raspberry by mean of an expansion board.

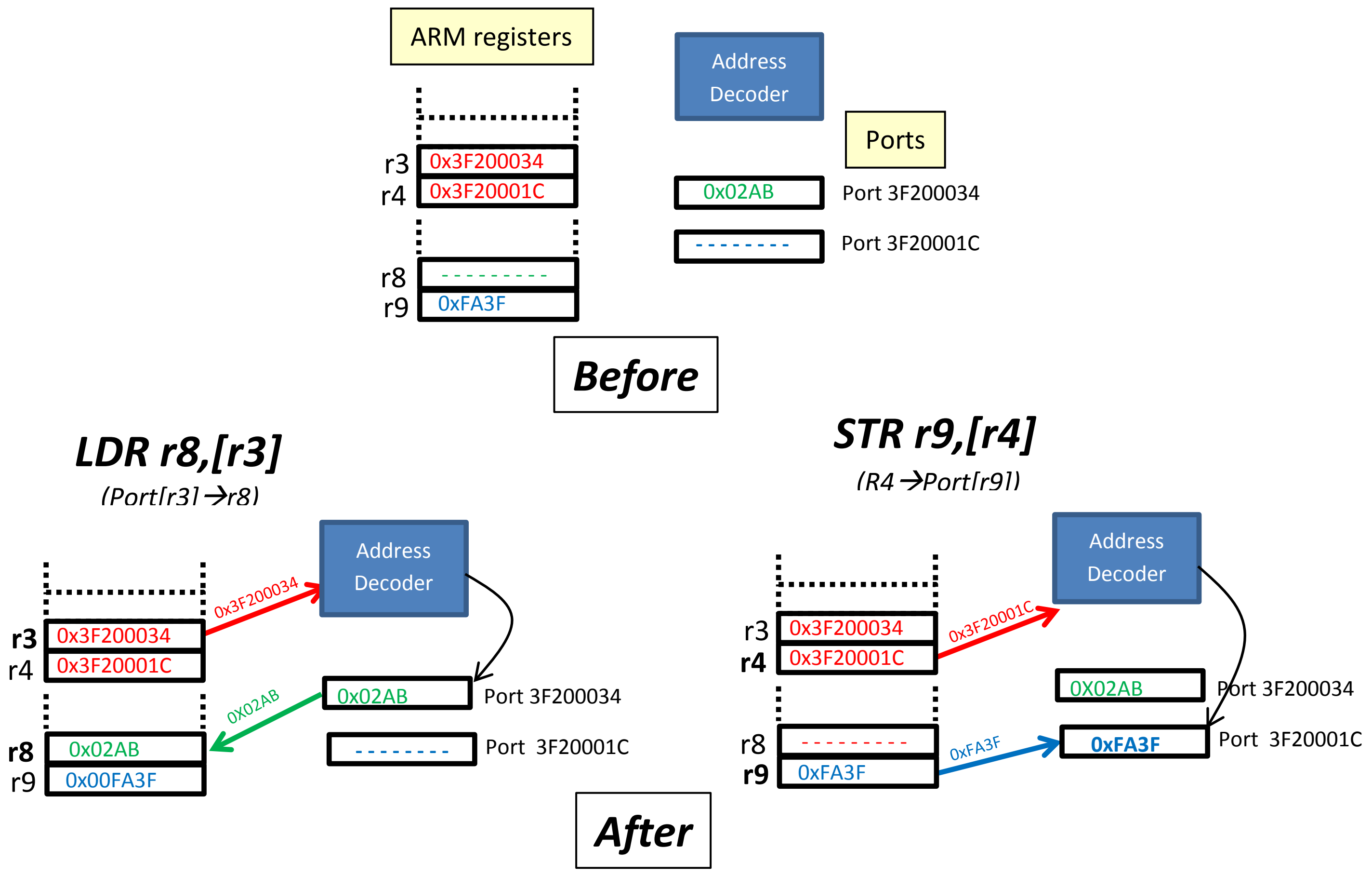
The way to communicate the ARM processor and the external devices is by using input/output ports. Each port has a unique address and is used for different purposes. We use the port system of the Raspberry to control the aforementioned devices, using the assembly language.

We assume that the students handle the ARM instruction set, including the fact that the instructions are conditional (a condition mnemonic is append to the instruction to indicate when to execute the instruction).

0.- Accessing I/O ARM ports

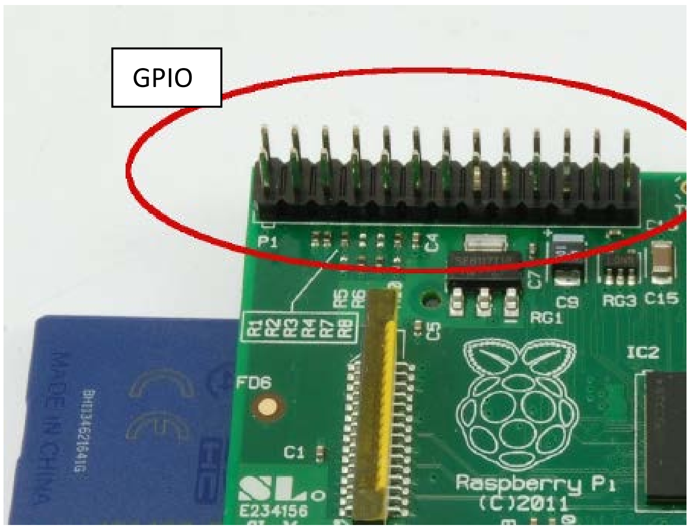
The input/output port system of the ARM processors is mapped with the memory. This means that access ports is carried out like access memory. The two instructions to access memory/IO are ***LDR*** and ***STR.*** Let us give some words about these instructions for I/O

- LDR is used to load an ARM register from an input port. The input port is defined by a specific address. A basic access input port is ***LDR r_x,[r_y]***, where ***r_x*** is the ARM register where the data will be loaded, and ***r_y*** is theARM register containing the address of the input port where the data is held.
- STR is used to copy an ARM register to an output port. A basic access input port is ***STR r_x,[r_y]***, where ***r_x*** is the ARM register that holds the data, and ***r_y*** is the ARM register containing the address of the output port (where the data will be sent).
  For example, if the address of an input port is 0x3F200034 and 0xF320001C is the address of an output port, , we have to load two ARM registers with these values before accessing the ports. Next figure shows an example

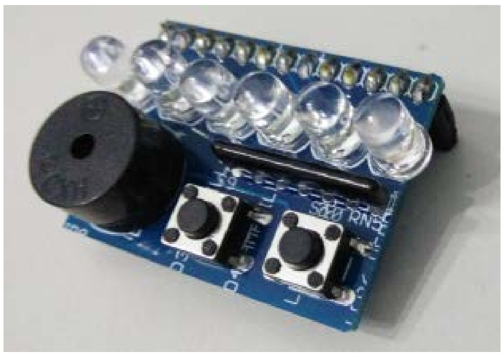


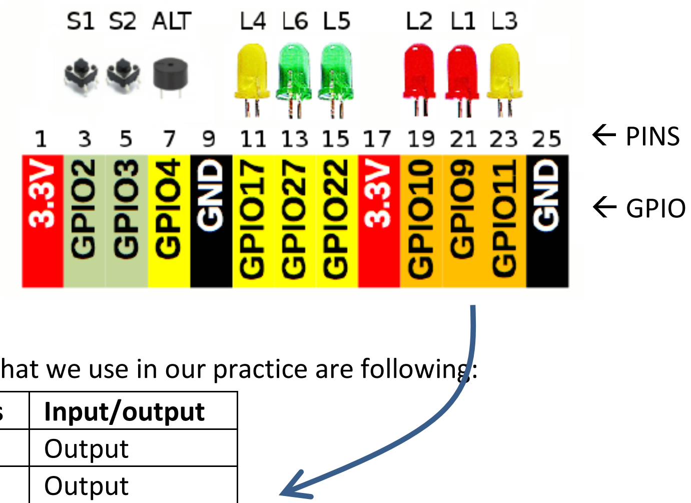
1.- GPIO (General-Purpose Input/Output)

GPIO is a set of 54 signals (namely GPIO0, GPIO1,…,GPIO53) for communication among the different parts of the Raspberry, which are connected to different ports. We will use a subset of these signals which are located in the connector P1. From now on, GPIO port denotes this connector, as shown in the next figure:



In our practice, we use only 9 out of the 54 signals of the GPIO which are located in the lower row of the connector. In turn, these 9 pins are connected to an expansion board which contents some devices, namely 6 colored Leds, 2 push buttons and one speaker. In the next figures, we can see the expansion board and number of GPIO signals associated to our 9 pins




S1 S2 ALT L4 L6 L5 L2 L1 L3
1 3 5 7 9 11 13 15 17 19 21 23 25
← PINS
3.3V GPIO2 GPIO3 GPIO4 GND GPIO17 GPIO27 GPIO22 3.3V GPIO10 GPIO9 GPIO11 GND
← GPIO

The 9 GPIO signals of the expansion board that we use in our practice are following:

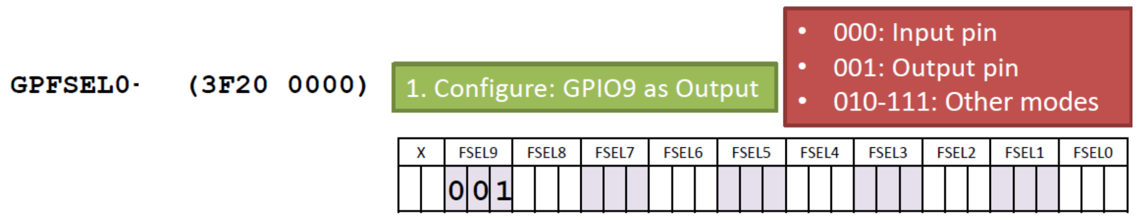
Device	GPIO	Board Pins	Input/output
LED1 (red)	9	21	Output
LED2 (red)	10	19	Output
LED3 (yellow)	11	23	Output
LED4 (yellow)	17	11	Output
LED5 (green)	22	15	Output
LED6 (green)	27	13	Output
PUSH_BUTTON1	2	3	Input
PUSH_BUTTON2	3	5	Input
SPEAKER	4	7	Output

1.1.- Configuring the GPIO pins as either input or output

The GPIO signals can be configured as individual input or output. Thus, before accessing the different devices on the board we have to configure them. In our case, we have 6 LED and one speaker connected to seven GPIO signals, which have to be configured as output, and two push buttons connected to 2 GPIO pins which have to be inputs.

The configuration of these signals as input/output is carried out by accessing some specific ports. To be exact, these ports are 3F 200 000 to configure GPIO0 through 9, 3F 200 004 to configure GPIO10 through 19 and 3F 200 008 for GPIO20 through 29. These ports have a code of three bits to configure each signal of the GPIO: the code **000** configure the corresponding signal as **input** and the code **001** configure it as **output**.

In the assembly compiler is possible to use symbolic names for the addresses. To be exact, the address **3F 200 000** is denoted **GPFSEL0**. Next figure show how to program the LED1 (which is associated to pin 9 of the GPIO) as an output:



According to the table of the section 1, we can see that

- LED1 & SPEAKER → GPIO 9 & 4 → Configured as output using the port 3F 200 000 with code 001
- PUSH BUTTONS → GPIO 2 & 3 → Configured as output using the port 3F 200 000 with code 000
- LED2 & 3 & 4 → GPIO 10 & 11 & 17→ Configured as output using the port 3F 200 004 with code 001
- LEDS5 & 6 → GPIO 22 & 27 → Configured as output using the port 3F 200 008 with code 001

Next table shows the code required for all the devices of our expansion board:

Address	Symbol			9			8			7			6			5			4			3			2			1			0		
3F200000	GPSEL0			0	0	1													0	0	1	0	0	0	0	0	0						
3F200004	GPSEL1									0	0	1																0	0	1	0	0	1
3F200008	GPSEL2									0	0	1													0	0	1						
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

As consequence, to work with our expansion board, we have to select these ports with the corresponding codes in the specific positions. To not modify the non-involved positions in the ports we have to use a mask to enforce “0” and “1” in the required positions only (see previous table). For example, the port 3F200008 (GPSEL2) has 4 bits enforced to “0” (bits 7, 8, 22 & 23) and two bits to “1” (bits 6 & 21), and the rest of the bits should not be modified. To force a “0” (and not modified the rest of the bits) a mask with an AND operation is required:

(1 111 111 $\overset{23}{0}\overset{22}{0}$1 111 … 111 $\overset{8}{0}\overset{7}{0}$1 111 111) **.AND.** (Port3F200008) → (Port3F200008)

and to enforce a “1” in bits 6 & 21 a mask with an OR operation is required:

(0000000000$\overset{21}{1}$00…$\overset{8}{0}$1000000)**.OR.** (Port 3F200008) → (Port3F200008).

Next code peforms the mask to force “0” at the positions 7,8,22&23 of Port 3F200008:

```
ldr r0, =0x3F200008
ldr r1,[r0]
ldr r2, =0b11111111001111111111111001111111
and r3,r1,r2
str r3,[r0]
```

Similarly, to enforce “1” the positions 6 & 21 of this port:

```
ldr r0, =0x3F200008
ldr r1,[r0]
ldr r2, =0b00000000000100000000000100000000
orr r3,r1,r2
str r3,[r0]
```

Therefore, this operation has to be carried out for the three ports 3F200000, 3F200004 & 3F200008. Next code is a good head of any code to manage the expansion board that is plugged in our Raspberry Pi2:

/* Programming the GPIO pins of the expansion board as input and output*/

```
ldr r0, =0x3F200000   /* r0=0x0x3F200000*/
/* Programming port 3F200000 (GPSEL0) */
ldr r1,[r0]                 /* r1<-Port(3F200000)
ldr r4, =0b11001111111111111001000000111111
ldr r5, =0b00000100000000000001000000000000
and r1,r1,r4
orr    r1,r1,r5
str r1,[r0]                 /* r1-> Port(3F200000)
/* Programming port 3F200004 (GPSEL1) */
ldr r1,[r0, #4]           /* r1<-Port(3F200004)
ldr r4, =0b11111111001111111111111111001001
ldr r5, =0b00000000001000000000000000001001
```

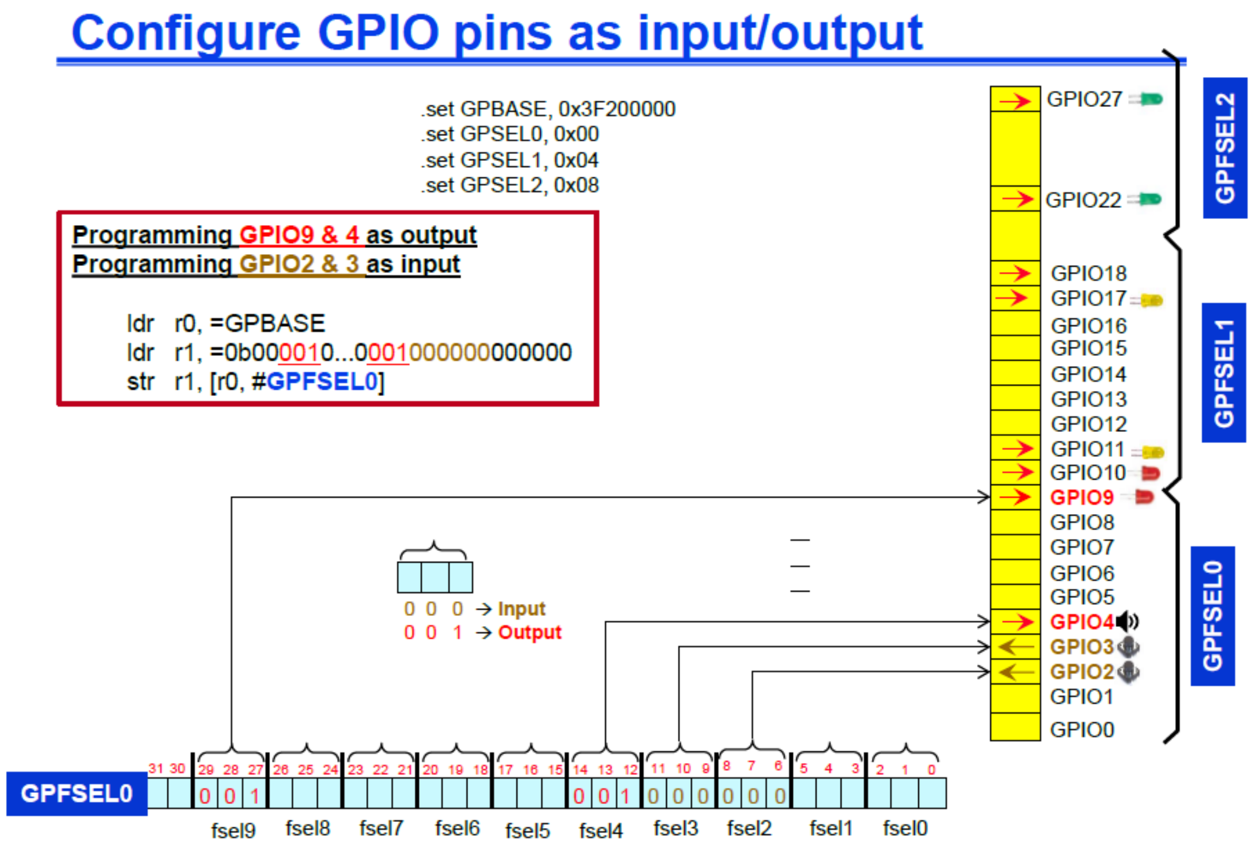
```
and r1,r1,r4
orr   r1,r1,r5
str r1, [r0, #4]               /* r1-> Port(3F200004) */
/* Programming port 3F200008 (GPSEL2) */
ldr r1,[r0, #8]                /* r1<-Port(3F200008) */
ldr r4, =0b11111111001111111111111001111111
ldr r5, =0b00000000001000000000000001000000
and r1,r1,r4
orr   r1,r1,r5
str r1,[r0, #8]                /* r1-> Port(3F200008) */
```

This code can be used as a head of any program since it has the code to configure all the signals associated to our expansion board. To make easy the edition of the program, it is advisable to create a ascii file "configuration.inc" and invoque it at the beginning of our program using the directive *.include "name_of_the_file"*; in our case, we have to write *.include "configuration.inc"*. In this file we can also see some directives like *.set* which make easy the edition of any program by using symbolic names. Check by yourself these and other directives.

configuration.inc

```
/* Configuration of all the I0 of the expansion board */
     .set   GPBASE,   0x3f200000
     .set   GPFSEL0,      0x00
     .set   GPFSEL1,      0x04
     .set   GPFSEL2,      0x08
.text
   ldr   r0, =GPBASE
         ldr   r1, [r0,  #GPFSEL0]
         ldr r4, =0b11001111111111111001000000111111 @ Mask for forcing 0
         ldr r5, =0b00001000000000000001000000000000 @ Mask for forcing 1
         and r1,r1,r4
         orr   r1,r1,r5
     str   r1, [r0, #GPFSEL0]                  @GPIO4&9 as output, GPIO2&3 as input
@ Configure of GPSEL1 (address 0x3F200004) for  GPIO 10,11,17
     ldr   r1, [r0, #GPFSEL1]
         ldr r4, =0b11111111001111111111111111001001  @ Mask for forcing 0
         ldr r5, =0b00000000001000000000000000001001  @ Mask for forcing 1
         and r1,r1,r4
         orr   r1,r1,r5
     str   r1, [r0, #GPFSEL1]                  @GPIO10&11&17 as output
@ Configure of GPSEL2 (address 0x3F200008) for  GPIO 22,27
     ldr   r1, [r0, #GPFSEL2]
         ldr r4, =0b11111111001111111111111001111111  @ Mask for forcing 0
         ldr r5, =0b00000000001000000000000001000000  @ Mask for forcing 1
         and r1,r1,r4
         orr   r1,r1,r5
     str   r1, [r0, #GPFSEL2]                  @GPIO22&27 as output
```

In the figure, we show an example of configuring the GPIO 9 & 4 as output and GPIO 2 & 3 as input:



Configure GPIO pins as input/output

```
.set GPBASE, 0x3F200000
.set GPSEL0, 0x00
.set GPSEL1, 0x04
.set GPSEL2, 0x08
```

Programming GPIO9 & 4 as output
Programming GPIO2 & 3 as input

```
ldr   r0, =GPBASE
ldr   r1, =0b000010...0001000000000000
str   r1, [r0, #GPFSEL0]
```

Symbolic Names

It is possible to use symbolic name to make easier the code writing. The file “***symbolic.inc***” contets them:

```
.macro    ADDEXC  vector, dirRTI
      ldr    r1, =(\dirRTI-\vector+0xa7fffffb)
      ror    r1, #2
      str    r1, [r0, #\vector]
    .endm
      .set   GPBASE,   0x3f200000
      .set   GPFSEL0,      0x00
      .set   GPFSEL1,      0x04
      .set   GPFSEL2,      0x08
      .set   GPFSEL3,      0x0c
      .set   GPFSEL4,      0x10
      .set   GPFSEL5,      0x14
      .set   GPFSEL6,      0x18
      .set   GPSET0,       0x1c
      .set   GPSET1,       0x20
      .set   GPCLR0,       0x28
      .set   GPCLR1,       0x2c
```

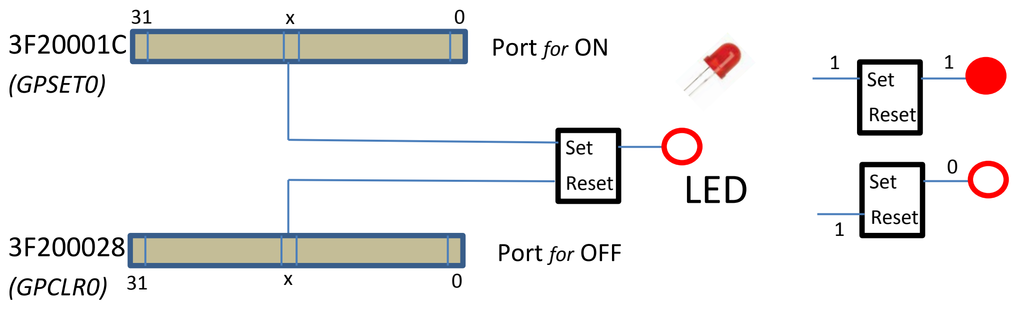
```
.set   GPLEV0,       0x34
.set   GPLEV1,       0x38
.set   GPEDS0,       0x40
.set   GPEDS1,       0x44
.set   GPFEN0,       0x58
.set   GPFEN1,       0x5c
.set   GPPUD,        0x94
.set   GPPUDCLK0,      0x98
.set   STBASE,   0x3f003000
.set   STCS,          0x00
.set   STCLO,          0x04
.set   STC1,          0x10
.set   STC3,          0x18
.set   INTBASE,  0x3f00b000
.set   INTFIQCON,    0x20c
.set   INTENIRQ1,    0x210
.set   INTENIRQ2,    0x214
```

1.2.- Accessing the expansion board

Once the GPIO signals of the expansion board are configured, we can access them. Our objective is to manage these devices using the input/output system of the ARM processor. The control of the different devices is as follows:

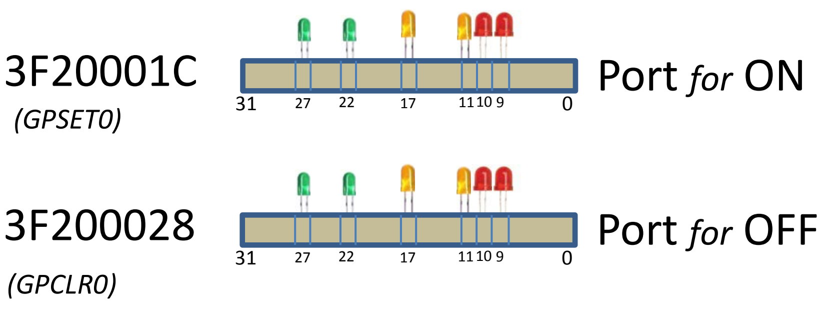
- **LEDs ON**: controlled by one bit of port 0x3F2000<u>1C</u>. If the suitable bit in this port is 1, the led will turn ON (after next clock edge).
- **LEDs OFF**: controlled by one bit of port 0x3F2000<u>28</u>. If the suitable bit in this port is 1, the led will turn OFF (after next clock edge).

  This means that we have to access different ports to turn ON and OFF each LED. The next logic circuit can help us to understand the functionality:



To turn ON the LED, we have to access port 3F20001C (with the bit x =1), and to turn OFF the same LED we access port 3F200028 (with the bit x =1).
The two red LEDs are connected to the bits 9 & 10 of the ports 3F20001C & 3F200028 respectively, the two yellow LEDs are with bits 11 & 17, and the two green LEDs are with bits 22 & 27.

3F20001C	ON					1					1					1					1	1	1										
3F200028	OFF					1					1					1					1	1	1										
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0



A very basic routine to turn ON the red LED of GPIO9 is

```
ldr  r0, =0x3F20001C     /*r0 contents the port address for ON  ldr  r0, =GPBASE */
ldr  r1, =0x200          /* r1= 0x20= 00…010 0000 0000  → bit9=1*/
str  r1,[r0]            /* LED ON: connect r1 with port  3F20001C, str r1,[r0, #GPSET0] */
```

Similarly, a very basic routine to turn OFF the red LED of GPIO9 is

```
        ldr  r0, =0x3F200028  /*r0 contents the port address for OFF, ldr  r0, =GPBASE */
        ldr  r1, =0x200            /* r1= 0x20= 00…010 0000 0000   → bit9=1*/
        str  r1,[r0]      /* LED OFF: connect r1 with port  3F200028,str r1,[r0, #GPSET0]    */
```

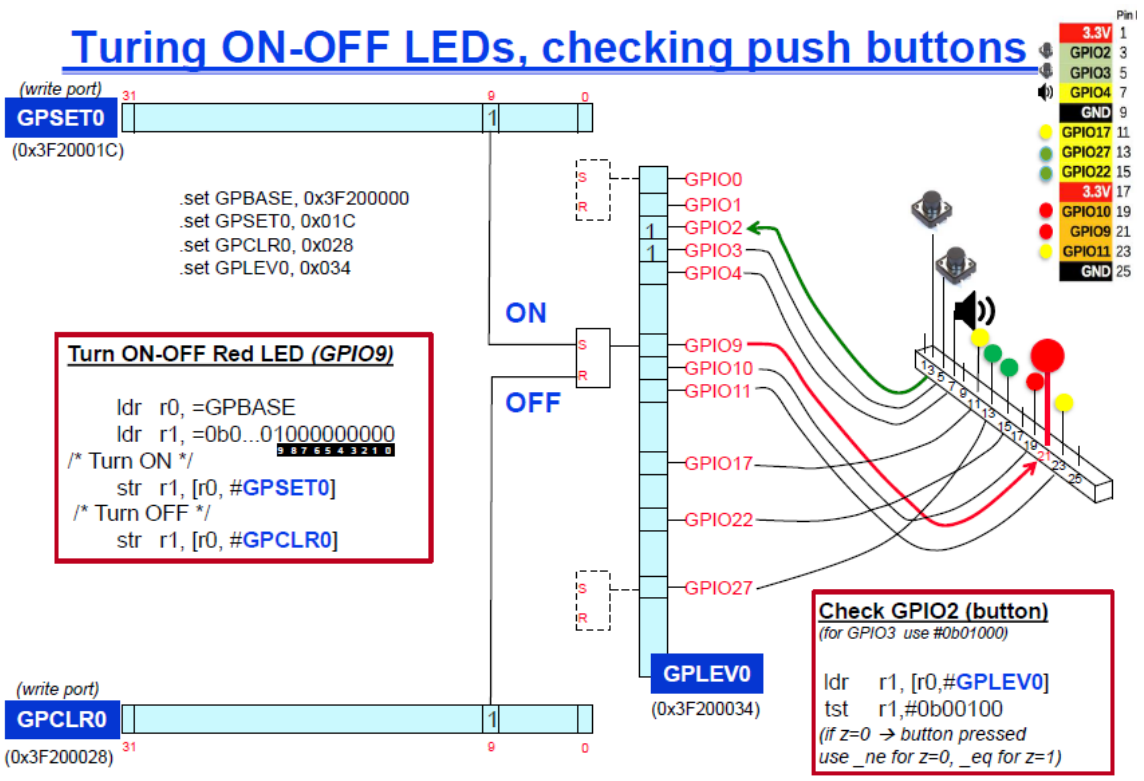
Now, it is time to write down our first program: design a program to ***turn on the red LED*** placed on GPIO9

```
        .include "configuration.inc"[1]
        ldr  r0, =0x3F20001C     /*r0 contents the port address for ON */
        ldr  r1, =0x200            /* r1= 0x200= 0000…0010 0000 0000   →bit9=1*/
        str  r1,[r0]               /* LED ON: connect r1 with port  3F20001C */
end:    b end[2]
```

Or you can also write:

```
        .include "configuration.inc"
        .include "symbolic.inc"[3]
        ldr  r0, =GPBASE           /*r0 contents the port base address for ON/OFF */
        ldr  r1, =0x200            /* r1= 0x200= 0000…0010 0000 0000   →bit9=1*/
        str  r1,[r0,#GPSET0]              /* LED ON: connect r1 with port  3F20001C */
end:    b end
```

This scheme summarizes and shows how to turn on/off a led



```
.set GPBASE, 0x3F200000
.set GPSET0, 0x01C
.set GPCLR0, 0x028
.set GPLEV0, 0x034
```

Turn ON-OFF Red LED *(GPIO9)*

```
      ldr   r0, =GPBASE
      ldr   r1, =0b0...01000000000
                      9 8 7 6 5 4 3 2 1 0
/* Turn ON */
      str   r1, [r0, #GPSET0]
 /* Turn OFF */
      str   r1, [r0, #GPCLR0]
```

Check GPIO2 (button)
(for GPIO3 use #0b01000)

```
 ldr     r1, [r0,#GPLEV0]
 tst     r1,#0b00100
```

(if z=0 → button pressed
use _ne for z=0, _eq for z=1)

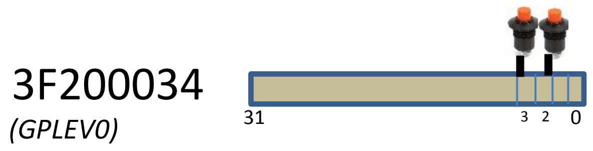
[1] The directive ".include" includes a file in the text. See "configuration.inc" in the previous page
[2] The program ends in this way due to the lack of an operating system.
[3] The directive ".include" includes a file in the text. See "symbolic.inc" in previous pages

Exercise 1: Write down a code to turn on one of the yellow leds and check it.

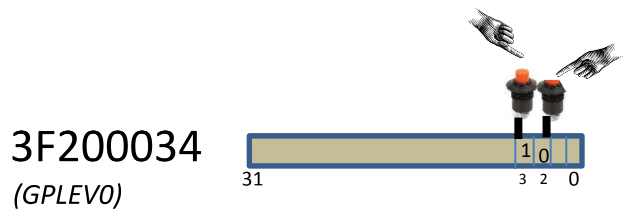
Exercise 2: Write down a code to turn on one of the green leds and check it.

Exercise 3: Write down a code to turn on one yellow led, one green led and one red led and check it.

- **PUSH BUTTONS**: they are connected to bits 2 & 3 of the port 3F2000034h.



If the button is pressed, a logic “0” is found in the corresponding bit of the port, otherwise a logic “1” is found:



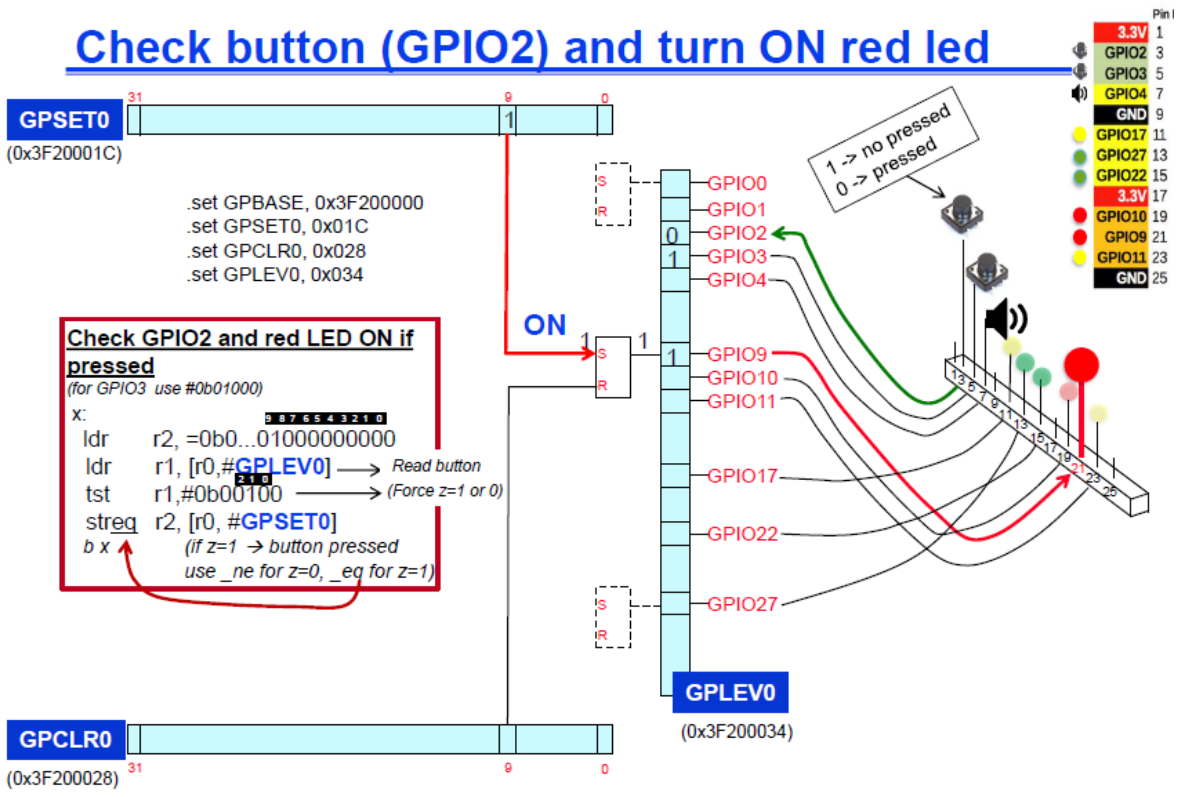
A very basic routine to copy the content of this port in the register r8 (for example) is:

```
ldr  r0,=0x3F200034     /* r0 contents the input port address, ldr  r0,=GPBASE */
ldr  r8,[r0]            /* port 3F200034 is copied into r8, ldr  r8,[r0,#GPLEV0]  */
                              /* If r8.2=0, the button is pressed (1 for released)*/
                              /* If r8.3=0, the button is pressed (1 for released)*/
```

There are several options to check one bit of a register. One of them is by mean of the instruction ***tst***. This instruction carries out the AND operation between a register and a specific mask and it modifies the ***z*** flag of the status register of the ARM processor. After the ***tst*** instruction, we can use the suffix ***_eq*** or ***_ne*** for the subsequent instruction(s) as needed. For example, to go to the routine “botton2pressed” if the push button is pressed, we can write the next code (assume we have a copy of the port *3F200034* in the register r8):

```
tst r8, #0b00100      /* The mask is 00100 for bit 2 */
beq button2pressed  /* if bit2=0 (pressed) → z=1 */
```

This scheme summarizes and shows an example of how to check a button to turn on a led



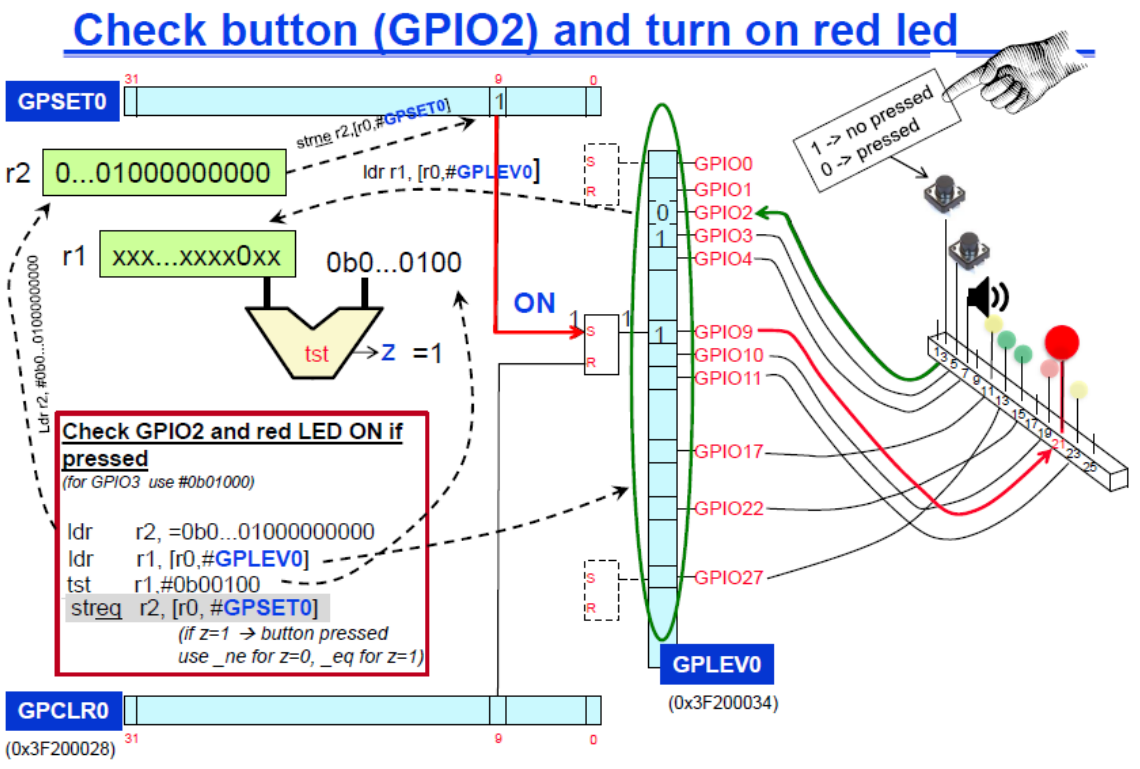
```
.set GPBASE, 0x3F200000
.set GPSET0, 0x01C
.set GPCLR0, 0x028
.set GPLEV0, 0x034
```

Check GPIO2 and red LED ON if pressed

(for GPIO3 use #0b01000)

```
x:
 ldr    r2, =0b0...01000000000
 ldr    r1, [r0,#GPLEV0]
 tst    r1,#0b00100
 streq  r2, [r0, #GPSET0]
 b x
```

(if z=1 → button pressed
use _ne for z=0, _eq for z=1)



Check GPIO2 and red LED ON if pressed

(for GPIO3 use #0b01000)

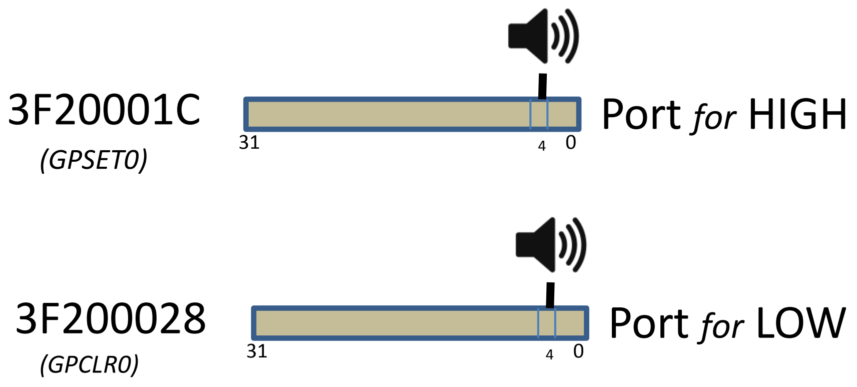
```
ldr    r2, =0b0...01000000000
ldr    r1, [r0,#GPLEV0]
tst    r1,#0b00100
streq  r2, [r0, #GPSET0]
```

(if z=1 → button pressed
use _ne for z=0, _eq for z=1)

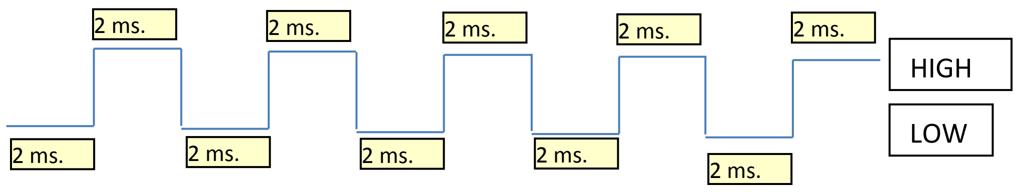
Now, do the next exercise

Exercise 4: Write down a code to turn on the yellow led on GPIO11 after pressing the push button

- **SPEAKER**: it is connected to the bit 4 of port 3F20001C. A sequence of 0-1-0-1-0… produces a wave which can be heard depending on the cadence 0-1-0-1-… selected. The cadence has to be such that the corresponding frequency belongs to the audible spectrum. As in the case of the LEDS, there are different ports for writing “1” or “0” and they are located at the same ports as LEDs: 3F20001C and 3F200028:



To produce a sound in the speaker we have to follow the sequence HIGH-LOW with a specific cadence. For example, for a sound of 250 Hz, we need a period of 1/250=0.004 s., that is, we need a HIGH for 0.002 s. (2 ms.) followed by a LOW for 0.002 s. in an infinite loop:



A very basic routine to produce a sound is:

```
        ldr  r0, =0x3F20001C     /*r0 contents the port address for HIGH */
        ldr  r2, =0x3F200028     /*r2 contents the port address for LOW */
        ldr  r1, =0x010          /* r1= 0x20= 00… 0001 0000   → bit4=1*/
loop:   str  r1,[r0]             /* HIGH */
        BL   wait                /* Routine for waiting 2 ms. */
        str  r1,[r2]             /* LOW */
        BL   wait                /* Routine for waiting 2 ms. */
        B    loop
```

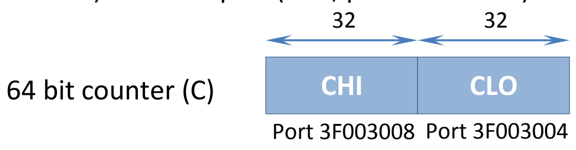
OR using symbolic manes (preferred):

```
        ldr  r0, =GPBASE         /*r0 contents the port base address for HIGH/LOW */
        ldr  r1, =0x010          /* r1= 0x20= 00… 0001 0000   → bit4=1*/
loop:   str  r1,[r0,#GPSET0]             /* HIGH */
        BL   wait                /* Routine for waiting 200 ms. */
        str  r1,[r0,#GPCLR0]             /* LOW */
        BL   wait                /* Routine for waiting 200 ms. */
        B    loop
```

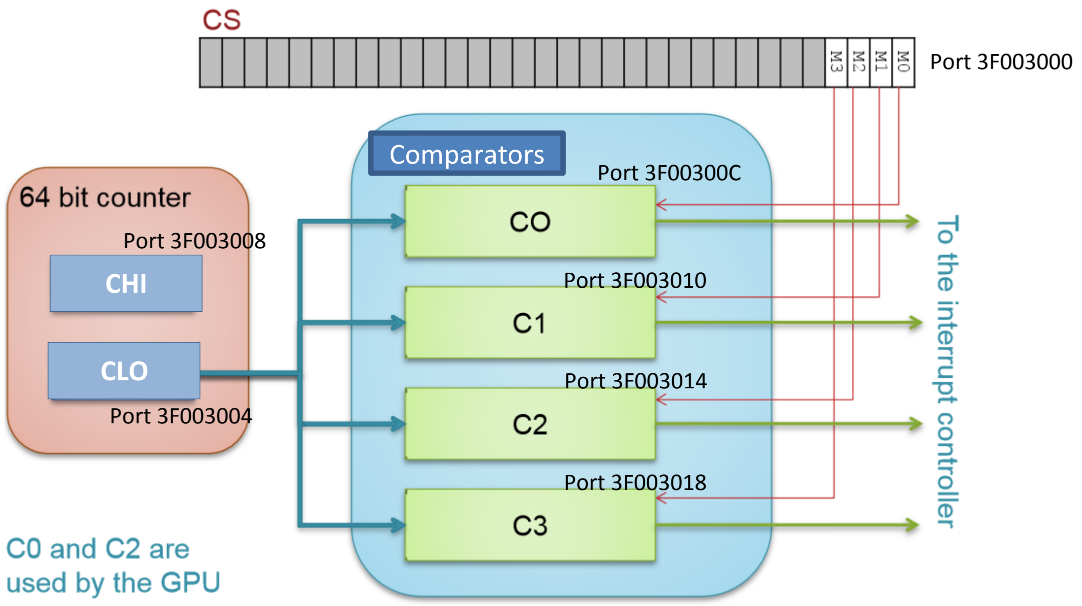
The routine for waiting 2ms (wait routine) will be generated from the ***timer***.

2.- TIMER

The Respberry Pi2 has a timer which is controlled by the ARM processor by mean of an input/output port system. It has a clock working at rate of 1 MHz (clock cycle of 1 µs), and increasing a counter (C) of 64 bits every 1 µs. The counter (C) is split into two parts: HIgh part (CHI, port 3F003008) and LOw part (CLO, port 3F003004).



A global scheme of the timer is



Apart from the 64 bit counter, the timer has 4 comparators (with 4 registers **C0,C1,C2** & **C3**) and one control/status register **CS**. Comparators are used to generate one interruption and will be studied later . All these registers are associated to input/output ports, as shown in the figure and summarized in the next table:

Timer register	Port Address
CS	3F003000
CLO	3F003004
CHI	3F003008
C0	3F00300C
C1	3F003010
C2	3F003014
C3	3F003018

The control of the time is very easy with the timer: just check the low part of the counter register CLO (port 3F003004) since we know it is incremented every µs[4].

[4] From a practical point of view, we only need to check the low part of the counter since the high part CHI is increased every 2^{32}µs= 1h11min

For example, to wait 2 ms = 2000 µs needed to generate a sound of 250 Hz we can read the register CLO and wait until this register has been increased by 2000 µs. Next code does that:

```
wait:   ldr r7, =0x3F003004 /* r7=0x3F003004 (address of counter CLO) , ldr r7,=STBASE*/
        ldr r3,[r7]         /* Read the value of the counter,  ldr r3,[r7,#STCLO */
        ldr r4, =2000 /* r4= 2 000 µs */
        add r4, r3, r4  /* Adding 2 000 µs to the current count to get the final count*/
ret1:   ldr r3,[r7]       /* Read the current count, ldr r3,[r7,#STCLO */
        cmp  r3,r4      /* Comparing current count with the final count */
        blt ret1
        bx   lr    /* returning to the main program after 2 000 µs*/
```

Now, try to do the next exercise:

Exercise 4: Write down a code to send out a continue tone of 250 Hz.

CAUTION: it is a good idea to use instructions PUSH and POP to deal with subroutine call so that the first instruction is a *push {registers_used}* and the penultimate is *pop {registers_used}.* To do that, you have to initialize the stack pointer of the supervisor mode (default mode after starting the Raspberry Pi 2) by inserting the next code at the beginning of your program (details will be shown in Part II of this manual):

```
/* Stack init for SVC mode  */
        mov     r0, #0b11010011
        msr    cpsr_c, r0
        mov     sp, #0x8000000
```

Thus, from now we use the next skeleton:

```
/* Basic skeleton for programs using ports (without interrupts) */
.include "configuration.inc"
.include "symbolic.inc"
/* Stack init for SVC mode         */
        mov    r0, #0b11010011
        msr    cpsr_c, r0
        mov    sp, #0x8000000
/* Continue my program here */


end:   b end
```

Exercise 5: Write down a code to send out a continue tone of 1000 Hz using push and pop

Exercise 6: Write a code to turn ON-OFF the two green leds with a cadence of 1 s (500 ms ON, 500 ms. OFF). Use push and pop

APPENDIX

Most frequently used ARM instructions for the practice (with examples)

<u>Memory Instructions:</u>

```
LDR r1, =0x012AF5        /* r1 = 0001 A2F5. Load a constant in a register
LDR r1, [r0]             /* P(r0) → r1. The port address is in r0
LDR r1, [r0, #4]         /* P(r0+4) → r1. The port address is  r0+4
STR r1, [r0]             /* P(r0) ← r1. The port address is in r0
STR r1, [r0, #4]          /* P(r0+4) ← r1. The port address is  r0+4
STReq  - -               /* like standard STR but it is executed only if flag Z=1 */
STRne  - -               /* like standard STR but it is executed only if flag Z=0 */
LDReq, STReq, LDRne, STRne  /* Similar but executed if Z=1/0
```

<u>Aritmetic/Logic:</u>

ADDS, SUBS, ANDS, ORRS, EORS → ADDSeq, SUBSeq, ANDSeq, ORRSeq, EXORSeq & _ne

<u>Branch Instructions</u>

```
B  xxx                   /* Inconditional branch to label xxx
CMP    r1,r2             /* Compare two registers
Bge, Bgt, Ble,Blt  /* Conditional branch depending on the result of the CMP instruction
Beq xxx          /* Conditional branch (branch if Z=1)
Bne xxx          /* Conditional branch (branch if Z=0)
BL  xxx          /* Branch and Link, function call a subroutine at label xxx
BX  LR           /* Return from a subroutine invoked by BL instruction
```

<u>Other instructions</u>

```
TST   r6,#0b00100        /* Test instruction: forces Z=1 if r6=xx..x0xx,  forces  Z=0 if r6=xx…x1xx

push {r1,r4}  /* copy r1 & r4 in the stack 4       r1,r4 → stack*/
pop  {r1,r4}  /* load r1 & r4 with the content of the stack   stack → r4,r1 */
```

Note: to use push and pop you have to initialize the stack pointer of the SVC mode by inserting the next code at the beginning of your program

```
/* Stack init for SVC mode  */
        mov    r0, #0b11010011
        msr    cpsr_c, r0
        mov    sp, #0x8000000
```

<u>Symbolic names file ("symbolic.inc")</u>

```
.macro    ADDEXC  vector, dirRTI
        ldr    r1, =(\dirRTI-\vector+0xa7fffffb)
        ror    r1, #2
        str    r1, [r0, #\vector]
      .endm
        .set   GPBASE,   0x3f200000
        .set   GPFSEL0,       0x00
        .set   GPFSEL1,       0x04
        .set   GPFSEL2,       0x08
        .set   GPFSEL3,       0x0c
        .set   GPFSEL4,       0x10
        .set   GPFSEL5,       0x14
        .set   GPFSEL6,       0x18
        .set   GPSET0,       0x1c
        .set   GPSET1,       0x20
        .set   GPCLR0,       0x28
        .set   GPCLR1,       0x2c
```

```
.set   GPLEV0,       0x34
.set   GPLEV1,       0x38
.set   GPEDS0,       0x40
.set   GPEDS1,       0x44
.set   GPFEN0,       0x58
.set   GPFEN1,       0x5c
.set   GPPUD,        0x94
.set   GPPUDCLK0,     0x98
.set   STBASE,   0x3f003000
.set   STCS,         0x00
.set   STCLO,         0x04
.set   STC1,         0x10
.set   STC3,         0x18
.set   INTBASE,  0x3f00b000
.set   INTFIQCON,    0x20c
.set   INTENIRQ1,    0x210
.set   INTENIRQ2,    0x214
```