Riccardo Beniamino 24 40 54

- **Es 3 Challange 1**
- **Es 3 Punti 4,3,2,1**

Inizio Esercitazione 3, Challenge 1

main.S:

```
/* Esercitazione 3, Challange 1
Cronometro che utilizza il TTC0 come base tempi
+ modifica base tempi con switch */
//button: 1 start, 2 stop, 3 edit, 4 reset
//swtich: 9 edit migliaia, 10 edit centinaia,
//11 edit decine, 12 edit unita'
//Novita' -> switch 1-4 divisione PRESCALER
//switch: 1 div massima->lento, 2 div media alta, 3 div media bassa, 4 div minima ->veloce
.text
.global main

@Declare functions as external
@This makes the labels acccessible in other functions
.extern ttc0_clk_conf
.extern ttc0_set_div
.extern ttc0_clk_setup
.extern ttc0_reset
.extern ttc0_en_itvl
.extern ttc0_set_itvl
.extern ttc0_en_itvl_int
.extern ttc0_init //quelle mie
.extern ttc0_delay_1ms
.extern set_dynamic_prescaler

.set SEG_CTL, 0x43C10000
.set SEG_DATA, 0x43C10004
.set SEG_DP, 0x43C10014

.set SW_DATA, 0x41220000
.set BTN_DATA, 0x41200000

main:
        bl ttc0_init //imposto tutti i parametri nel ttc0
    mov R0, #0  //Inizializza la cifra delle migliaia a 0
    mov R1, #0  //Inizializza la cifra delle centinaia a 0
    mov R2, #0  //Inizializza la cifra delle decine a 0
    mov R3, #0  //Inizializza la cifra delle unità a 0
    bl seg_enable

loop:
  //bl ttc0_delay_1ms
  bl set_dynamic_prescaler
  bl ttc0_delay_1ms
  bl button_switch_logic
  bl digits_data_adapter
  bl seg_print
b loop

button_switch_logic:
        push {r4,r5,r6,r7,lr}
        ldr r4,=BTN_DATA
        ldr r5,[r4] //su r5 ho il valore dei button

        cmp r5, #1 //quarto bottone reset
        bne exit1
        mov r0, #0
        mov r1, #0
        mov r2, #0
    mov r3, #0
        exit1:

        cmp r5,#2 //terzo bottone edit
        bne exit2

        a:
        ldr r5,[r4] //rileggo il pulsante
        cmp r5,#0 //fronte di discesa
```

```
        bne a   //ciclo finche non vedo il fronte di discesa

    ldr r6,=SW_DATA
    ldr r7,[r6] //su r7 ho il valore degli switch
    and r7,r7,#0x00F //mi interessano solo i primi 4 switch

        cmp r7,#1 //se i sw sono a 1 incremnto unita' (ultimo sw on)
        bne b
        adds r3,r3,#1
    b:

        cmp r7,#2 //se i sw sono a 2 incremento decine (penultimo sw on)
        bne c
        adds r2,r2,#1
    c:

    cmp r7,#4 //se i sw sono a 4 incremento le centinaia (terzultimo sw on)
    bne d
    adds r1,r1,#1
    d:

    cmp r7,#8 //se i sw sono a 8 incremento le migliaia (quartultimo sw on)
    bne e
    adds r0,r0,#1
    e:

        //bl simple_counter

        exit2:
        ldr r5,[r4]
    cmp r5, #8 //se il primo button e' premuto: start
    bne f
    mov r8,#1
    f:

    cmp r5, #4 //se il secondo button e' premuto: stop
    bne g
    mov r8, #0
    g:

    cmp r8, #1
    bne h
    add r3,r3,#1
    h:

    bl simple_counter
        pop {r4,r5,r6,r7,lr}
bx lr

simple_counter:
    //add R3, R3, #1   //Incrementa la cifra delle unità
    cmp R3, #10      //Controlla se la cifra delle unità supera 9
    bne check_decine //Se R4 < 10, salta a check_tens
    mov R3, #0       //Altrimenti, resetta le unità a 0
    add R2, R2, #1   //Incrementa la cifra delle decine

  check_decine:
    cmp R2, #10         //Controlla se la cifra delle decine supera 9
    bne check_centinaia //Se R3 < 10, salta a check_hundreds
    mov R2, #0          //Altrimenti, resetta le decine a 0
    add R1, R1, #1      //Incrementa la cifra delle centinaia

  check_centinaia:
    cmp R1, #10         //Controlla se la cifra delle centinaia supera 9
    bne check_migliaia  //Se R2 < 10, salta a check_thousands
    mov R1, #0          //Altrimenti, resetta le centinaia a 0
    add R0, R0, #1      //Incrementa la cifra delle migliaia

  check_migliaia:
    cmp R0, #10 //Controlla se la cifra delle migliaia supera 9
    bne exit    //Se R1 < 10, esci, aggiorna il display
            //Se R1 raggiunge 10, abbiamo superato 9999, quindi possiamo resettare tutto
    mov R0, #0  //Resetta le migliaia a 0
    mov R1, #0  //Resetta le centinaia a 0
    mov R2, #0  //Resetta le decine a 0
    mov R3, #0  //Resetta le unità a 0
```

```
    exit: //esco,aggiorno il display
bx lr

digits_data_adapter:
//R0 = migliaia, R1 = centinaia, R2 = decinie, R1 = unita'
//Colloco opprtunamente queste cifre nel display
//i dati sono passati in uscita su r4, per andare sul display
   push {r0,r1,r2,r3}
   lsl r0,r0,#24 //shift a sx di 24 per arrivare nei BCD_3 data
   lsl r1,r1,#16 //shift a sx di 16 per arrivare nei BCD_2 data
   lsl r2,r2,#8  //shift a sx di 8 per arrivare nei BCD_1 data
           //le unita' sono gia' al BCD_0 data
   mov r4, #0
   adds r4,r0
   adds r4,r1
   adds r4,r2
   adds r4,r3

   pop {r0,r1,r2,r3}
bx lr

seg_enable:
  push {r0,r1}
  ldr r0,=SEG_CTL
  mov r1, #1 //bit 0, 1 -> modalita' BCD di default, enable
  str r1,[r0]
  pop {r0,r1}
bx lr

seg_print: //ricevo su r4 i valori da stampare a schermo
  push {r0,r1,r4}
  ldr r0,=SEG_DATA
  ldr r1,=0x80808080 //maschera per spegnere i DP
  orr r4,r4,r1     //fatta con una OR bitwise
  str r4,[r0]
  pop {r0,r1,r4}
bx lr
```

ttc0.S:

```
@Declare functions as global
@This makes the labels acccessible in other functions
.global ttc0_clk_conf
.global ttc0_set_div
.global ttc0_clk_setup
.global ttc0_reset
.global ttc0_en_itvl
.global ttc0_set_itvl
.global ttc0_en_itvl_int
.global ttc0_init //quelle mie
.global ttc0_delay_1ms
.global set_dynamic_prescaler
@ Addresses
.equ TTC0_BASE, 0xF8001000 @ ttc0 clock control reg
.equ TTC0_CLKCTL_OFF, 0x0
.equ TTC0_CNTCTL_OFF, 0xC @ ttc0 count 1 control
.equ TTC0_ITVL_OFF, 0x24 @ ttc0 interval 1 reg
.equ TTC0_IER_OFF, 0x60 @ ttc0 IER (interrupt enable)
.equ TTC0_ISR_OFF, 0x54 @ ttc0 ISR (interrupt status)
.set SW_DATA, 0x41220000

@ Constants used to setup timer
.equ div_mask, 0xFFFFFFE1 @ mask to clear only bits 4:1
.equ cnt_dis_rst, 0x11 @ settings to disable and reset counter
.equ cnt_en_itvl, 0x22 @ settings to enable counter in interval mode
.equ intvl_int_en, 0x1 @ settings to enable interrupt signal so it can be read

@ configures TTC0 to use prescale and pclk
@ sets div_value to 0 (div pclk by 2)
ttc0_clk_conf:
        ldr r1, =(TTC0_BASE+TTC0_CLKCTL_OFF) @ load address of clock control reg
        mov r0, #1
        str r0, [r1]
bx lr
```

```
@ sets the clock divider for
@ takes r0 as 4-bit prescale value
@ sets bits 4:1 in the TTC clock control register
ttc0_set_div:
        ldr r2, =TTC0_BASE
        ldr r1, [r2] @ get current conf
        ldr r3, =div_mask @ load mask
        and r1, r3, r1 @ mask out div bits
        and r0, r0, #0xF @ mask out all but 4 LSB
        lsl r0,r0, #1 @ shift left one (align div bits)
        orr r1,r0, r1 @ combine r0,r1
        str r1, [r2] @ write new conf
bx lr

@ NOTE: The above two code segments illustrate how individual
@ bit fields in control registers can be set without inadvertently
@ changing bits in neighboring fields. In this project, you can
@set
@ the seven bits in the CLK_CNTRL register to "0010011" to enable
@prescale,
@ set a prescale divide constant of 9, and select the positive
@edge of the PC
@ clock as the clock source. The code below does this in a
@simplified fashion.
ttc0_clk_setup:
        ldr r1, =TTC0_BASE @ load TTC0 base register address into R1
        mov r0, #0b0010011 @ r0 <- bit settings to divide PC clock by 2^10
        str r0, [r1] @ store bits into TTC0 base register
bx lr


@ resets and disables TTC0 counter 1
@ the counter must be reenabled manually
ttc0_reset:
        ldr r1, =(TTC0_BASE + TTC0_CNTCTL_OFF)
        mov r0, #cnt_dis_rst @ assert reset
        str r0,[r1] @ store constant
bx lr

@ sets TTC0 counter 1 as enabled in interval mode
ttc0_en_itvl:
        ldr r1, =(TTC0_BASE + TTC0_CNTCTL_OFF)
        mov r0, #cnt_en_itvl @ enable interval mode
        str r0, [r1]
bx lr

@ takes in interval value (16-bit) as parameter
@ in r0, writes to itvl register
ttc0_set_itvl:
        ldr r1,=(TTC0_BASE+TTC0_ITVL_OFF)
        str r0, [r1] @ store value
bx lr

@ enables ttc0 interval interrupts
@ no parameter
ttc0_en_itvl_int:
        ldr r1, =(TTC0_BASE+TTC0_IER_OFF)
        mov r0, #intvl_int_en
        str r0, [r1]
bx lr

//quelle mie ////////////////////////////////////////////////////////////
ttc0_init:
        push {lr}
    bl ttc0_reset    //"Disabilita", forse intendeva RESETTA il contatore
    bl ttc0_clk_conf //Configura il clock del timer

    mov r0, #0 //Punto BONUS //9 //Setta il valore del prescaler a 9
    bl ttc0_set_div //Prescaler value (può essere cambiato in base alle esigenze)

    mov r0, #0xFFFF //=65535 BONUS: sfrutto tutti i 16 bit del registro
    bl ttc0_set_itvl //Interval value (può essere cambiato in base alle esigenze)

    bl ttc0_en_itvl_int //5. Abilita interval interrupt flag

    bl ttc0_en_itvl //6. Abilita il timer in interval mode
```

```
        pop {lr}
bx lr

//funzione per implementare il delay di 1ms utilizzando l'interrupt polling del TTC0
ttc0_delay_1ms:
    push {r0,r1,lr} //salvo il lr sullo stack, dato che contiene l'indirizzo di ritorno, cosi la funzione
puo' tornare al chiamante

    polling_loop:
    ldr r1, =(TTC0_BASE + TTC0_ISR_OFF) //carico su r1 l'inidirzzo dell'InterruptStateRegister
    ldr r0, [r1] //carico il valore del registro ISR nel registro r0
    tst r0, #1   //Testa il bit 0 di r0, utilizzando l'istr. TST (test), che esegue un'operazione AND tra r0
e 1 (maschera per il bit 0) senza modificare r0. Questo imposta i flag di condizione basati sul risultato
    beq polling_loop //Se il bit 0 non e' impostato (cioe' se r0 AND 1 e' zero), esegue un branch ritornando
a 'polling_loop' per ripetere il ciclo di polling

    pop {r0,r1,lr} //dopo che il bit 0 e' stato impostato, il ciclo di polling termina.
bx lr              //il lr viene ripristinato dallo stack e poi si esegue un branch all'indirizzo contenuto
nel link register per ritornare dalla subroutine

// Modifica dinamica della base temporale del timer
set_dynamic_prescaler: //Non funziona bene, da controllare
    push {r0, r1, r2, r3, lr}
    ldr r1, =SW_DATA

    mov r2, #0
    ldr r2, [r1]   // Legge lo stato degli switch
    and r2, r2, #0xF00 // Maschera per prendere solo i primi 4 bit

    // Utilizzare i valori degli switch per determinare il valore del prescaler
    cmp r2, #256
    bne switch_case_1
    mov r0, #0   // 1 KHz default a 0 oppure con il 4 switch on
    bl ttc0_set_div
    b end_set_dynamic_prescaler

switch_case_1:
    cmp r2, #512
    bne switch_case_2
    mov r0, #2  // 500 Hz quando il 3 switch on
    bl ttc0_set_div
    b end_set_dynamic_prescaler

switch_case_2:
    cmp r2, #1024
    bne switch_case_3
    mov r0, #4  // 250 Hz quando il 2 switch on
    bl ttc0_set_div
    b end_set_dynamic_prescaler

switch_case_3:
    cmp r2, #2048
    bne switch_case_4
    mov r0, #8 // 125 Hz quando il 1 swtich on
    bl ttc0_set_div
    b end_set_dynamic_prescaler

switch_case_4:
    // Aggiungere ulteriori casi se necessario
    b end_set_dynamic_prescaler

end_set_dynamic_prescaler:
    pop {r0, r1, r2, r3, lr}
    bx lr
```

Fine Esercitazione 3, Challenge 1

Inizio Esercitazione 3, punto 4
main.S

```
/* Esercitazione 3, punto 4
Cronometro che utilizza il TTC0 come base tempi */
//button: 1 start, 2 stop, 3 edit, 4 reset
//swtich: 9 edit migliaia, 10 edit centinaia,
//11 edit decine, 12 edit unita'
.text
.global main

@Declare functions as external
@This makes the labels acccessible in other functions
.extern ttc0_clk_conf
.extern ttc0_set_div
.extern ttc0_clk_setup
.extern ttc0_reset
.extern ttc0_en_itvl
.extern ttc0_set_itvl
.extern ttc0_en_itvl_int
.extern ttc0_init //quelle mie
.extern ttc0_delay_1ms

.set SEG_CTL, 0x43C10000
.set SEG_DATA, 0x43C10004
.set SEG_DP, 0x43C10014

.set SW_DATA, 0x41220000
.set BTN_DATA, 0x41200000

main:
        bl ttc0_init //imposto tutti i parametri nel ttc0
    mov R0, #0  //Inizializza la cifra delle migliaia a 0
    mov R1, #0  //Inizializza la cifra delle centinaia a 0
    mov R2, #0  //Inizializza la cifra delle decine a 0
    mov R3, #0  //Inizializza la cifra delle unità a 0
    bl seg_enable

loop:
  bl ttc0_delay_1ms
  bl button_switch_logic
  bl digits_data_adapter
  bl seg_print
b loop

button_switch_logic:
        push {r4,r5,r6,r7,lr}
        ldr r4,=BTN_DATA
        ldr r5,[r4] //su r5 ho il valore dei button

        cmp r5, #1 //quarto bottone reset
        bne exit1
        mov r0, #0
        mov r1, #0
        mov r2, #0
    mov r3, #0
        exit1:

        cmp r5,#2 //terzo bottone edit
        bne exit2

        a:
        ldr r5,[r4] //rileggo il pulsante
        cmp r5,#0 //fronte di discesa
        bne a   //ciclo finche non vedo il fronte di discesa

    ldr r6,=SW_DATA
    ldr r7,[r6] //su r7 ho il valore degli switch
    and r7,r7,#0x00F //mi interessano solo i primi 4 switch
```

```
        cmp r7,#1 //se i sw sono a 1 incremnto unita' (ultimo sw on)
        bne b
        adds r3,r3,#1
    b:

        cmp r7,#2 //se i sw sono a 2 incremento decine (penultimo sw on)
        bne c
        adds r2,r2,#1
    c:

    cmp r7,#4 //se i sw sono a 4 incremento le centinaia (terzultimo sw on)
    bne d
    adds r1,r1,#1
    d:

    cmp r7,#8 //se i sw sono a 8 incremento le migliaia (quartultimo sw on)
    bne e
    adds r0,r0,#1
    e:

        //bl simple_counter

        exit2:
        ldr r5,[r4]
    cmp r5, #8 //se il primo button e' premuto: start
    bne f
    mov r8,#1
    f:

    cmp r5, #4 //se il secondo button e' premuto: stop
    bne g
    mov r8, #0
    g:

    cmp r8, #1
    bne h
    add r3,r3,#1
    h:

    bl simple_counter
        pop {r4,r5,r6,r7,lr}
bx lr

simple_counter:
    //add R3, R3, #1   //Incrementa la cifra delle unità
    cmp R3, #10      //Controlla se la cifra delle unità supera 9
    bne check_decine //Se R4 < 10, salta a check_tens
    mov R3, #0       //Altrimenti, resetta le unità a 0
    add R2, R2, #1   //Incrementa la cifra delle decine

  check_decine:
    cmp R2, #10         //Controlla se la cifra delle decine supera 9
    bne check_centinaia //Se R3 < 10, salta a check_hundreds
    mov R2, #0          //Altrimenti, resetta le decine a 0
    add R1, R1, #1      //Incrementa la cifra delle centinaia

  check_centinaia:
    cmp R1, #10         //Controlla se la cifra delle centinaia supera 9
    bne check_migliaia  //Se R2 < 10, salta a check_thousands
    mov R1, #0          //Altrimenti, resetta le centinaia a 0
    add R0, R0, #1      //Incrementa la cifra delle migliaia

  check_migliaia:
    cmp R0, #10 //Controlla se la cifra delle migliaia supera 9
    bne exit    //Se R1 < 10, esci, aggiorna il display
            //Se R1 raggiunge 10, abbiamo superato 9999, quindi possiamo resettare tutto
    mov R0, #0  //Resetta le migliaia a 0
    mov R1, #0  //Resetta le centinaia a 0
    mov R2, #0  //Resetta le decine a 0
    mov R3, #0  //Resetta le unità a 0

    exit: //esco,aggiorno il display
bx lr

digits_data_adapter:
//R0 = migliaia, R1 = centinaia, R2 = decinie, R1 = unita'
```

```
//Colloco opprtunamente queste cifre nel display
//i dati sono passati in uscita su r4, per andare sul display
   push {r0,r1,r2,r3}
   lsl r0,r0,#24 //shift a sx di 24 per arrivare nei BCD_3 data
   lsl r1,r1,#16 //shift a sx di 16 per arrivare nei BCD_2 data
   lsl r2,r2,#8  //shift a sx di 8 per arrivare nei BCD_1 data
          //le unita' sono gia' al BCD_0 data
   mov r4, #0
   adds r4,r0
   adds r4,r1
   adds r4,r2
   adds r4,r3

   pop {r0,r1,r2,r3}
bx lr

seg_enable:
  push {r0,r1}
  ldr r0,=SEG_CTL
  mov r1, #1 //bit 0, 1 -> modalita' BCD di default, enable
  str r1,[r0]
  pop {r0,r1}
bx lr

seg_print: //ricevo su r4 i valori da stampare a schermo
  push {r0,r1,r4}
  ldr r0,=SEG_DATA
  ldr r1,=0x80808080 //maschera per spegnere i DP
  orr r4,r4,r1     //fatta con una OR bitwise
  str r4,[r0]
  pop {r0,r1,r4}
bx lr
```

ttc0.S

```
@Declare functions as global
@This makes the labels acccessible in other functions
.global ttc0_clk_conf
.global ttc0_set_div
.global ttc0_clk_setup
.global ttc0_reset
.global ttc0_en_itvl
.global ttc0_set_itvl
.global ttc0_en_itvl_int
.global ttc0_init //quelle mie
.global ttc0_delay_1ms

@ Addresses
.equ TTC0_BASE, 0xF8001000 @ ttc0 clock control reg
.equ TTC0_CLKCTL_OFF, 0x0
.equ TTC0_CNTCTL_OFF, 0xC @ ttc0 count 1 control
.equ TTC0_ITVL_OFF, 0x24 @ ttc0 interval 1 reg
.equ TTC0_IER_OFF, 0x60 @ ttc0 IER (interrupt enable)
.equ TTC0_ISR_OFF, 0x54 @ ttc0 ISR (interrupt status)

@ Constants used to setup timer
.equ div_mask, 0xFFFFFFE1 @ mask to clear only bits 4:1
.equ cnt_dis_rst, 0x11 @ settings to disable and reset counter
.equ cnt_en_itvl, 0x22 @ settings to enable counter in interval mode
.equ intvl_int_en, 0x1 @ settings to enable interrupt signal so it can be read

@ configures TTC0 to use prescale and pclk
@ sets div_value to 0 (div pclk by 2)
ttc0_clk_conf:
        ldr r1, =(TTC0_BASE+TTC0_CLKCTL_OFF) @ load address of clock control reg
        mov r0, #1
        str r0, [r1]
bx lr

@ sets the clock divider for
@ takes r0 as 4-bit prescale value
@ sets bits 4:1 in the TTC clock control register
ttc0_set_div:
        ldr r2, =TTC0_BASE
        ldr r1, [r2] @ get current conf
        ldr r3, =div_mask @ load mask
        and r1, r3, r1 @ mask out div bits
```

```
        and r0, r0, #0xF @ mask out all but 4 LSB
        lsl r0,r0, #1 @ shift left one (align div bits)
        orr r1,r0, r1 @ combine r0,r1
        str r1, [r2] @ write new conf
bx lr

@ NOTE: The above two code segments illustrate how individual
@ bit fields in control registers can be set without inadvertently
@ changing bits in neighboring fields. In this project, you can
@set
@ the seven bits in the CLK_CNTRL register to "0010011" to enable
@prescale,
@ set a prescale divide constant of 9, and select the positive
@edge of the PC
@ clock as the clock source. The code below does this in a
@simplified fashion.
ttc0_clk_setup:
        ldr r1, =TTC0_BASE @ load TTC0 base register address into R1
        mov r0, #0b0010011 @ r0 <- bit settings to divide PC clock by 2^10
        str r0, [r1] @ store bits into TTC0 base register
bx lr


@ resets and disables TTC0 counter 1
@ the counter must be reenabled manually
ttc0_reset:
        ldr r1, =(TTC0_BASE + TTC0_CNTCTL_OFF)
        mov r0, #cnt_dis_rst @ assert reset
        str r0,[r1] @ store constant
bx lr

@ sets TTC0 counter 1 as enabled in interval mode
ttc0_en_itvl:
        ldr r1, =(TTC0_BASE + TTC0_CNTCTL_OFF)
        mov r0, #cnt_en_itvl @ enable interval mode
        str r0, [r1]
bx lr

@ takes in interval value (16-bit) as parameter
@ in r0, writes to itvl register
ttc0_set_itvl:
        ldr r1,=(TTC0_BASE+TTC0_ITVL_OFF)
        str r0, [r1] @ store value
bx lr

@ enables ttc0 interval interrupts
@ no parameter
ttc0_en_itvl_int:
        ldr r1, =(TTC0_BASE+TTC0_IER_OFF)
        mov r0, #intvl_int_en
        str r0, [r1]
bx lr

//quelle mie ////////////////////////////////////////////////////////////
ttc0_init:
        push {lr}
    bl ttc0_reset    //"Disabilita", forse intendeva RESETTA il contatore
    bl ttc0_clk_conf //Configura il clock del timer

    mov r0, #0 //Punto BONUS //9 //Setta il valore del prescaler a 9
    bl ttc0_set_div //Prescaler value (può essere cambiato in base alle esigenze)

    mov r0, #65535 //BONUS: sfrutto tutti i 16 bit del registro //108 // 4. Setta l'intervallo a 108 conteggi
    bl ttc0_set_itvl //Interval value (può essere cambiato in base alle esigenze)

    bl ttc0_en_itvl_int //5. Abilita interval interrupt flag

    bl ttc0_en_itvl //6. Abilita il timer in interval mode
        pop {lr}
bx lr

//funzione per implementare il delay di 1ms utilizzando l'interrupt polling del TTC0
ttc0_delay_1ms:
    push {r0,r1,lr} //salvo il lr sullo stack, dato che contiene l'indirizzo di ritorno, cosi la funzione
puo' tornare al chiamante
```

```
    polling_loop:
    ldr r1, =(TTC0_BASE + TTC0_ISR_OFF) //carico su r1 l'inidirzzo dell'InterruptStateRegister
    ldr r0, [r1] //carico il valore del registro ISR nel registro r0
    tst r0, #1   //Testa il bit 0 di r0, utilizzando l'istr. TST (test), che esegue un'operazione AND tra r0
e 1 (maschera per il bit 0) senza modificare r0. Questo imposta i flag di condizione basati sul risultato
    beq polling_loop //Se il bit 0 non e' impostato (cioe' se r0 AND 1 e' zero), esegue un branch ritornando
a 'polling_loop' per ripetere il ciclo di polling

    pop {r0,r1,lr} //dopo che il bit 0 e' stato impostato, il ciclo di polling termina.
bx lr              //il lr viene ripristinato dallo stack e poi si esegue un branch all'indirizzo contenuto
nel link register per ritornare dalla subroutine
```

Fine Esercitazione 3, punto 4

Spiegazione su Esercitazione 3, punto 4 "BONUS"

"**Bonus:** puoi ottenere una base temporale ancora più precisa! Usa un po' di matematica e trova una combinazione di valore di intervallo e divisore che ti avvicini il più possibile a 1 ms. Per ottenere la massima precisione, è necessario che il clock sia il più veloce possibile, ma la lunghezza dell'intervallo è limitata alla dimensione del registro (16 bit). Il valore dell'intervallo più grande è quindi 65535 ((2^16 == 65536) -1 == 65535"

Per ottenere una base temporale ancora più precisa, pari a 1ms, calcolo una combinazione di valore di intervallo e divisore che si avvicini il più possibile a 1ms. Il timer TTC ha un clock periferico a 111 MHz, che possiamo dividere usando un valore di prescaler.

Calcolo dei parametri

Per ottenere una precisione massima, utilizzeremo il valore massimo di intervallo 65535, in esadecimale 0xFFFF e troveremo il valore di prescaler che ci permette di avvicinarci il più possibile a un intervallo di 1 ms.

La formula per il tempo dell'intervallo è:

$$Tempo_{intervallo} = \frac{interval}{pclk} \times 2^{(prescale+\ )}$$

Dove: pclk=111 Mhz, interval=65535, prescale = ?

Risoluzione

Per ottenere un intervallo di 1 ms (0.001 secondi), dobbiamo risolvere la seguente equazione per "prescale":

$$0.001 \approx \frac{65535}{111 \times 10^6} \times 2^{(prescale+1)}$$

Calcolo

Scomponiamo l'equazione:

$$0.001 \times 111 \times 10^6 \approx 65535 \times 2^{(prescale+1)}$$
$$111000 \approx 65535 \times 2^{(prescale+\ )}$$
$$\frac{111000}{65535} \approx 2^{(prescale+1)}$$
$$1.694 \approx 2^{(prescale+1)}$$

Prendendo il logaritmo in base 2 su entrambi i lati otteniamo:

$$log_2(1.694) \approx prescale + 1$$
$$0.765 \approx prescale + 1$$
$$prescale \approx -0.235$$

Poiché il valore di prescale deve essere un numero intero positivo, arrotondiamo al valore intero piu' vicino. Quindi prescale sarà 0.

Inizio Esercitazione 3, punto 3
main.S

```
/* Esercitazione 3, punto 3
Progetta un flusso di controllo per il tuo cronometro
Modifica il tuo programma per aggiungere le
funzioni Start, Stop, Incremento e Clear */

//button: 1 start, 2 stop, 3 edit, 4 reset
//swtich: 9 edit migliaia, 10 edit centinaia,
//11 edit decine, 12 edit unita'
.text
.global main

.set SEG_CTL, 0x43C10000
.set SEG_DATA, 0x43C10004
.set SEG_DP, 0x43C10014

.set SW_DATA, 0x41220000
.set BTN_DATA, 0x41200000

main:
    mov R0, #0  //Inizializza la cifra delle migliaia a 0
    mov R1, #0  //Inizializza la cifra delle centinaia a 0
    mov R2, #0  //Inizializza la cifra delle decine a 0
    mov R3, #0  //Inizializza la cifra delle unità a 0
    bl seg_enable

loop:
  bl constant_delay //antirimbalzo
  bl button_switch_logic
  //bl simple_counter
  bl digits_data_adapter
  bl seg_print
b loop

button_switch_logic:
        push {r4,r5,r6,r7,lr}
        ldr r4,=BTN_DATA
        ldr r5,[r4] //su r5 ho il valore dei button

        cmp r5, #1 //quarto bottone reset
        bne exit1
        mov r0, #0
        mov r1, #0
        mov r2, #0
    mov r3, #0
        exit1:

        cmp r5,#2 //terzo bottone edit
        bne exit2

        a:
        ldr r5,[r4] //rileggo il pulsante
        cmp r5,#0 //fronte di discesa
        bne a   //ciclo finche non vedo il fronte di discesa

    ldr r6,=SW_DATA
    ldr r7,[r6] //su r7 ho il valore degli switch
    and r7,r7,#0x00F //mi interessano solo i primi 4 switch

        cmp r7,#1 //se i sw sono a 1 incremnto unita' (ultimo sw on)
        bne b
        adds r3,r3,#1
    b:

        cmp r7,#2 //se i sw sono a 2 incremento decine (penultimo sw on)
        bne c
```

```
        adds r2,r2,#1
    c:

    cmp r7,#4 //se i sw sono a 4 incremento le centinaia (terzultimo sw on)
    bne d
    adds r1,r1,#1
    d:

    cmp r7,#8 //se i sw sono a 8 incremento le migliaia (quartultimo sw on)
    bne e
    adds r0,r0,#1
    e:

        //bl simple_counter

        exit2:
        ldr r5,[r4]
    cmp r5, #8 //se il primo button e' premuto: start
    bne f
    mov r8,#1
    f:

    cmp r5, #4 //se il secondo button e' premuto: stop
    bne g
    mov r8, #0
    g:

    cmp r8, #1
    bne h
    add r3,r3,#1
    h:

    bl simple_counter
        pop {r4,r5,r6,r7,lr}
bx lr

simple_counter:
    //add R3, R3, #1   //Incrementa la cifra delle unità
    cmp R3, #10      //Controlla se la cifra delle unità supera 9
    bne check_decine //Se R4 < 10, salta a check_tens
    mov R3, #0       //Altrimenti, resetta le unità a 0
    add R2, R2, #1   //Incrementa la cifra delle decine

  check_decine:
    cmp R2, #10         //Controlla se la cifra delle decine supera 9
    bne check_centinaia //Se R3 < 10, salta a check_hundreds
    mov R2, #0          //Altrimenti, resetta le decine a 0
    add R1, R1, #1      //Incrementa la cifra delle centinaia

  check_centinaia:
    cmp R1, #10         //Controlla se la cifra delle centinaia supera 9
    bne check_migliaia  //Se R2 < 10, salta a check_thousands
    mov R1, #0          //Altrimenti, resetta le centinaia a 0
    add R0, R0, #1      //Incrementa la cifra delle migliaia

  check_migliaia:
    cmp R0, #10 //Controlla se la cifra delle migliaia supera 9
    bne exit    //Se R1 < 10, esci, aggiorna il display
            //Se R1 raggiunge 10, abbiamo superato 9999, quindi possiamo resettare tutto
    mov R0, #0  //Resetta le migliaia a 0
    mov R1, #0  //Resetta le centinaia a 0
    mov R2, #0  //Resetta le decine a 0
    mov R3, #0  //Resetta le unità a 0

    exit: //esco,aggiorno il display
bx lr

digits_data_adapter:
//R0 = migliaia, R1 = centinaia, R2 = decinie, R1 = unita'
//Colloco opprtunamente queste cifre nel display
//i dati sono passati in uscita su r4, per andare sul display
   push {r0,r1,r2,r3}
   lsl r0,r0,#24 //shift a sx di 24 per arrivare nei BCD_3 data
   lsl r1,r1,#16 //shift a sx di 16 per arrivare nei BCD_2 data
   lsl r2,r2,#8  //shift a sx di 8 per arrivare nei BCD_1 data
           //le unita' sono gia' al BCD_0 data
```

```
   mov r4, #0
   adds r4,r0
   adds r4,r1
   adds r4,r2
   adds r4,r3

   pop {r0,r1,r2,r3}
bx lr

seg_enable:
  push {r0,r1}
  ldr r0,=SEG_CTL
  mov r1, #1 //bit 0, 1 -> modalita' BCD di default, enable
  str r1,[r0]
  pop {r0,r1}
bx lr

seg_print: //ricevo su r4 i valori da stampare a schermo
  push {r0,r1,r4}
  ldr r0,=SEG_DATA
  ldr r1,=0x80808080 //maschera per spegnere i DP
  orr r4,r4,r1     //fatta con una OR bitwise
  str r4,[r0]
  pop {r0,r1,r4}
bx lr

constant_delay:
  push {r0}
    ldr r0, = 0xA2990 //666 mln = 1 sec, 666 mln/1000 = 1 mS = 666'000
  delay_loop:
      subs r0, r0, #1   // Decrementa il conteggio
      bne delay_loop    // Ripeti finché il conteggio non è zero
  pop {r0}
bx lr             // Ritorna dalla subroutine
```

Fine Esercitazione 3, punto 3

Inizio Esercitazione 3, punto 2

```
//Esercitazione 3, punto 2
//pulsante 3 incrementa cifra, pulsante 4 reset
//il gruppo di 4 switch a destra seleziona il digit da editare
//es: se l'ultimo sw e' on incremento le unita'
//se il penultimo sw e' on incremnto le decine ecc...
//con incremento opportuno del counter
.text
.global main

.set SEG_CTL, 0x43C10000
.set SEG_DATA, 0x43C10004
.set SEG_DP, 0x43C10014

.set SW_DATA, 0x41220000
.set BTN_DATA, 0x41200000

main:
    mov R0, #0  //Inizializza la cifra delle migliaia a 0
    mov R1, #0  //Inizializza la cifra delle centinaia a 0
    mov R2, #0  //Inizializza la cifra delle decine a 0
    mov R3, #0  //Inizializza la cifra delle unità a 0
    bl seg_enable

loop:
  bl constant_delay //antirimbalzo
  bl button_switch_logic
  //bl simple_counter
  bl digits_data_adapter
  bl seg_print
b loop

button_switch_logic:
        push {r4,r5,r6,r7,lr}
        ldr r4,=BTN_DATA
        ldr r5,[r4] //su r5 ho il valore dei button
```

```
        cmp r5, #1 //quarto bottone reset
        bne exit1
        mov r0, #0
        mov r1, #0
        mov r2, #0
    mov r3, #0
        exit1:

        cmp r5,#2 //terzo bottone edit
        bne exit2

        a:
        ldr r5,[r4] //rileggo il pulsante
        cmp r5,#0 //fronte di discesa
        bne a   //ciclo finche non vedo il fronte di discesa

    ldr r6,=SW_DATA
    ldr r7,[r6] //su r7 ho il valore degli switch
    and r7,r7,#0x00F //mi interessano solo i primi 4 switch

        cmp r7,#1 //se i sw sono a 1 incremnto unita' (ultimo sw on)
        bne b
        adds r3,r3,#1
    b:

        cmp r7,#2 //se i sw sono a 2 incremento decine (penultimo sw on)
        bne c
        adds r2,r2,#1
    c:

    cmp r7,#4 //se i sw sono a 4 incremento le centinaia (terzultimo sw on)
    bne d
    adds r1,r1,#1
    d:

    cmp r7,#8 //se i sw sono a 8 incremento le migliaia (quartultimo sw on)
    bne e
    adds r0,r0,#1
    e:

        bl simple_counter

        exit2:
        pop {r4,r5,r6,r7,lr}
bx lr

simple_counter:
    //add R3, R3, #1   //Incrementa la cifra delle unità
    cmp R3, #10      //Controlla se la cifra delle unità supera 9
    bne check_decine //Se R4 < 10, salta a check_tens
    mov R3, #0       //Altrimenti, resetta le unità a 0
    add R2, R2, #1   //Incrementa la cifra delle decine

  check_decine:
    cmp R2, #10         //Controlla se la cifra delle decine supera 9
    bne check_centinaia //Se R3 < 10, salta a check_hundreds
    mov R2, #0          //Altrimenti, resetta le decine a 0
    add R1, R1, #1      //Incrementa la cifra delle centinaia

  check_centinaia:
    cmp R1, #10         //Controlla se la cifra delle centinaia supera 9
    bne check_migliaia  //Se R2 < 10, salta a check_thousands
    mov R1, #0          //Altrimenti, resetta le centinaia a 0
    add R0, R0, #1      //Incrementa la cifra delle migliaia

  check_migliaia:
    cmp R0, #10 //Controlla se la cifra delle migliaia supera 9
    bne exit    //Se R1 < 10, esci, aggiorna il display
            //Se R1 raggiunge 10, abbiamo superato 9999, quindi possiamo resettare tutto
    mov R0, #0  //Resetta le migliaia a 0
    mov R1, #0  //Resetta le centinaia a 0
    mov R2, #0  //Resetta le decine a 0
    mov R3, #0  //Resetta le unità a 0

    exit: //esco,aggiorno il display
bx lr
```

```
digits_data_adapter:
//R0 = migliaia, R1 = centinaia, R2 = decinie, R1 = unita'
//Colloco opprtunamente queste cifre nel display
//i dati sono passati in uscita su r4, per andare sul display
   push {r0,r1,r2,r3}
   lsl r0,r0,#24 //shift a sx di 24 per arrivare nei BCD_3 data
   lsl r1,r1,#16 //shift a sx di 16 per arrivare nei BCD_2 data
   lsl r2,r2,#8  //shift a sx di 8 per arrivare nei BCD_1 data
           //le unita' sono gia' al BCD_0 data
   mov r4, #0
   adds r4,r0
   adds r4,r1
   adds r4,r2
   adds r4,r3

   pop {r0,r1,r2,r3}
bx lr

seg_enable:
  push {r0,r1}
  ldr r0,=SEG_CTL
  mov r1, #1 //bit 0, 1 -> modalita' BCD di default, enable
  str r1,[r0]
  pop {r0,r1}
bx lr

seg_print: //ricevo su r4 i valori da stampare a schermo
  push {r0,r1,r4}
  ldr r0,=SEG_DATA
  ldr r1,=0x80808080 //maschera per spegnere i DP
  orr r4,r4,r1     //fatta con una OR bitwise
  str r4,[r0]
  pop {r0,r1,r4}
bx lr

constant_delay:
  push {r0}
    ldr r0, = 0x186A0 //= 100'000
  delay_loop:
      subs r0, r0, #1   // Decrementa il conteggio
      bne delay_loop    // Ripeti finché il conteggio non è zero
  pop {r0}
bx lr             // Ritorna dalla subroutine
```

Fine Esercitazione 3, punto 2

Inizio Esercitazione 3, punto 1

```
//Esercitazione 3, punto 1
/* Usa le tue funzioni per creare un programma che legga un numero a 12 bit dagli
interruttori e un numero a 4 bit dai pulsanti. Visualizza il numero dagli interruttori sulle
prime tre cifre del display a 7 segmenti (considera ciascun gruppo di quattro interruttori
come una cifra esadecimale) e visualizza il numero letto dai pulsanti sulla quarta cifra */
.text
.global main

.set SEG_CTL, 0x43C10000
.set SEG_DATA, 0x43C10004
.set SEG_DP, 0x43C10014

.set SW_DATA, 0x41220000
.set BTN_DATA, 0x41200000

main:
bl seg_enable

loop:
        bl read_switch
        bl read_button
        bl seg_print
b loop
```

```
seg_enable:
        push {r0,r1}
        ldr r0,=SEG_CTL
        mov r1, #1 //bit 0, 1 -> modalita' BCD di default, enable
        str r1,[r0]
        pop {r0,r1}
bx lr

read_switch:            //ritorna su r1 il valore letto dagli switch
    push {r0,r2,r3} //r1: valore a 12 bit, contiene:
        ldr r0,=SW_DATA //3 sottoinsiemi da 4 bit gia' opportunamente mascherati e shiftati
        ldr r1,[r0]     //alla fine sommo tutto e riottengo 12 bit nell'ordine giusto
                                    //per pilotare i primi 3 digit del display
        mov r2,r1 //copio r1, su r2, appoggio
        mov r3,r1 //copio r1, su r3, appoggio

        and r1,r1,#0xF00  //tengo solo i 4 bit piu' signif. dei 12
        lsl r1,r1,#16 //shift a sx di 16 per arrivare nei BCD_3 data

        and r2,r2,#0x0F0 //tengo solo i 4 bit centrali dei 12 (bit 8..5)
        lsl r2,r2,#12 //shift a sx di 12 per arrivare nei BCD_2 data

        adds r1,r1,r2

        and r3,r3,#0x00F //tengo solo i 4 bit meno signif. dei 12
        lsl r3,r3,#8 //shift a sx di 8 per arrivare nei BCD_1 data

        adds r1,r1,r3
        pop {r0,r2,r3}
bx lr

read_button:  //somma su r1 il valore letto dai button
    push {r0,r2} //valore a 4 bit
        ldr r0,=BTN_DATA
        ldr r2,[r0]
        adds r1,r1,r2 //sommo a r1 i 4 bit gia' nella pos meno sigificativa
        pop {r0,r2}   //sono gia' nella posiz. giusta per BCD_0 Data
bx lr

seg_print: //ricevo su r1 i valori da stampare a schermo
        push {r0,r1,r2}
        ldr r0,=SEG_DATA
        ldr r2,=0x80808080 //maschera per spegnere i DP
        orr r1,r1,r2       //fatta con una OR bitwise
        str r1,[r0]
        pop {r0,r1,r2}
bx lr
```

Fine Esercitazione 3, punto 1