INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Arquitetura de Computadores

4º Trabalho Prático

Trabalho realizado por:

Nome:	Francisco Lufinha	Nº 49447
Nome:	Pedro Malafaia	Nº 49506
Nome:	Roberto Petrisoru	Nº 49418

Docente: Rui Policarpo Duarte

Índice

1. Introdução

Este relatório, referente ao quarto trabalho prático da cadeira Arquitetura de Computadores, irá dar resposta às perguntas formuladas no enunciado, bem como irá ter toda a informação considerada relevante para a compreensão deste trabalho.

Este projeto teve como desafio o desenvolvimento em linguagem *assembly* do jogo Simple e-Craps. Este trata-se de uma versão simplificada de Craps. Resumidamente, o objetivo é um jogador tentar acertar num número gerado pseudo aleatoriamente, entre 0 e 15.

Para observar o programa é correr, é necessário utilizar uma placa SDP16, uma placa ATB, um circuito Pico Timer (pTC) e um mostrador de 7 segmentos. A solução adotada para ligar as placas SDP16 e ATB ao circuito será apresentada mais à frente neste relatório.

2. Elementos relevantes para compreensão do trabalho

Tal como foi referido anteriormente, para pôr em prática o programa desenvolvido em *assembly*, de modo a ser possível jogar o jogo, é necessário a ligação do mostrador de 7 segmentos à placa SDP16, bem como a ligação do pTC à placa SDP16 e à placa ATB.

O mostrador de 7 segmentos será responsável por, nada mais nada menos, do que apresentar visualmente a aposta do jogador, a simulação do lançamento do dado e, posteriormente, o número que foi gerado aleatoriamente. É importante referir que este mostrador pisca quando o jogador acerta no número.

Já o circuito pTC é responsável por contar ciclos de um sinal gerado externamente. Neste caso, o circuito pTC recebe um sinal de relógio da placa ATB de 1Khz. A contagem deste sinal de relógio, permite-nos obter os intervalos de tempo necessários para a realização deste programa como, por exemplo, a contagem dos 10 segundos, após o número aleatório ter sido gerado.

3. Resposta às questões

1. *"Apresente a solução adotada para ligar o circuito pTC à placa SDP16".*

A solução adotada para ligar o circuito pTC à placa SDP16 e à placa ATB é possível observar na figura 1.

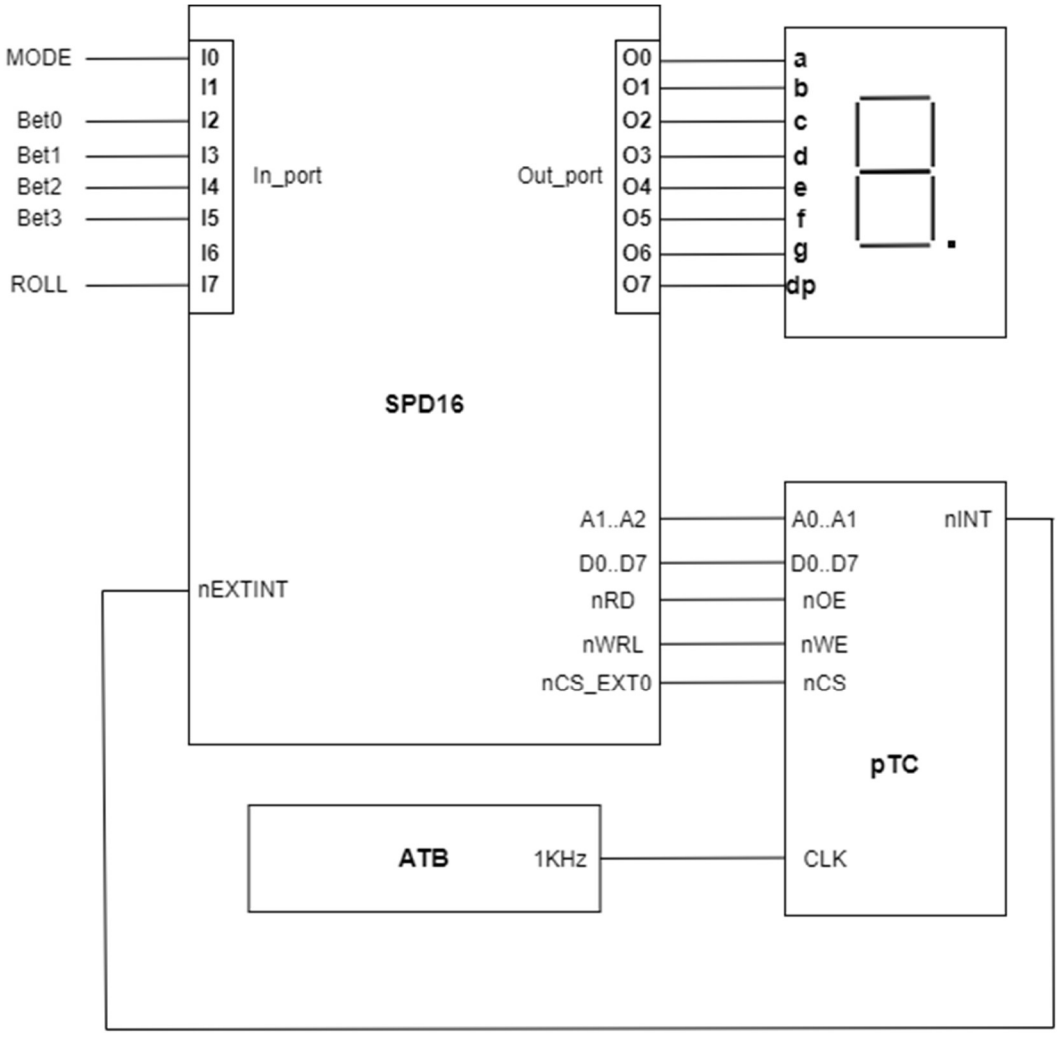

Figura 1 – Solução adotada para a ligação do circuito pTC às placas ATB e SPD16

2. *"Explique os cálculos realizados para determinar as temporizações envolvidas neste trabalho."*

De maneira que rotina de interrupção seja chamada com um intervalo de 100ms, colocou-se o TMR com o valor 99 (0x63) uma vez que depois de se dar reset, o 0 que fica também conta. Assim de 100 em 100 ms a rotina ISR incrementa o valor da variável sysclk. Todas as outras rotinas utilizam esta variável de maneira a controlarem o tempo que necessário dependendo da sua função.

3. *"Indique, justificando, a latência máxima do sistema no atendimento dos pedidos de interrupção gerados pelo circuito pTC."*

A latência máxima são 0,18ms, pois é o tempo que demora a ser executada uma instrução em que são necessários 6 ciclos de clk. Uma vez que o sistema só atende um pedido de interrupção quando não estiver a executar nenhuma instrução, assim a latência máxima ocorre quando o sistema recebe um pedido de interrupção imediatamente apos começar a executar uma instrução com 6 ciclos de clk.

4. *"Indique, justificando, quanto tempo demora, no pior caso, a execução da rotina utilizada para o atendimento da interrupção externa."*

(X * 3) + (Y * 6) = ? ciclos de clk

X corresponde ao número de instruções que não acedem à memória, neste caso X = 14

Y corresponde ao número de instruções que acedem à memória (ldr, str, pop, push), neste caso Y = 5.

Neste caso, ficamos com (14 * 3) + (5 * 6), que é corresponde a 72 ciclos de clk.

Como a SDP16 tem uma frequência de 50kHz, 72 ciclos de clk demoram 1,44 ms.

4. Listagem do programa realizado

Nesta fase do relatório iremos colocar o programa realizado em *assembly*, devidamente identado e comentado. É importante referir que foram realizadas algumas alterações em relação ao código entregue na data prevista. As alterações realizadas foram as seguintes:

- No modo de aposta, em que o jogador escolhe o número que pretende, o código entregue na data prevista não ligava o ponto no mostrador de 7 segmentos, indicando que estaríamos no modo de aposta.
- Após a transição descendente, terminando assim o modo de aposta, não permitindo ao jogador a partir daí alterar o seu número escolhido, o código entregue na data prevista simplesmente apagava o mostrador, até que fosse detetado a transição ascendente do switch I7, iniciando assim o lançamento do dado. A alteração realizada foi a permanência no mostrador do número escolhido pelo jogador, mas com o ponto desligado, indicando assim que a aposta já foi realizada e não pode ser revertida.

De seguida, irá ser possível observar todo o código em *assembly*.

```
;; CONSTANTS

    .equ    SEV_SEG_OFF, 0x0        ; seven segment off
    .equ    ROLL_EFFECT_COUNT, 0x04 ; roll effect count
    .equ    ANIMATION_SIZE, 0x06    ; display_roulette_array size
    .equ    STACK_SIZE, 128         ; stack size = 128 B

    ; max random number (4 bits)
    .equ RAND_MAX_L, 0x0F
    .equ RAND_MAX_H, 0x00

    ; possible FSM states
    .equ    STATE_WAIT_FOR_MODE, 0
    .equ    STATE_GET_BET, 1
    .equ    STATE_WAIT_FOR_ROLL, 2
    .equ    STATE_ROLL_EFFECT, 3
    .equ    STATE_ROLL_DICE, 4
    .equ    STATE_CHECK_WIN, 5
    .equ    STATE_FINAL_TIMER, 6

    ; masks
```

```
    .equ    BET_MASK, 0x3C          ; 0b00111100 (I5..2)
    .equ    MODE_MASK, 0x1          ; 0b00000001 (I0)
    .equ    ROLL_MASK, 0x80         ; 0b10000000 (I7)
    .equ    RAND_MASK, 0xF          ; rand() 4 bit mask
    .equ    CPSR_BIT_I, 0b010000    ; CPSR I mask
    .equ    DOT_ON, 0b10000000      ; dot on sevseg

    ; timer settings
    .equ    PTC_ADDRESS,  0xFF40          ; pTC address
    .equ    PTC_TCR, 0                    ; pTC TCR's offset index
    .equ    PTC_TMR, 2                    ; pTC TMR's offset index
    .equ    PTC_TC, 4                     ; pTC TC's offset index
    .equ    PTC_TIR, 6                    ; pTC TIR's offset index
    .equ    PTC_CMD_START, 0              ; pTC start command
    .equ    PTC_CMD_STOP, 1               ; pTC stop command

    .equ    SYSCLK_FREQ, 0x63               ; interval to set in TMR
    .equ    TEN_SECS, 0x64                  ; 100 * 100 ms = 10 sec
    .equ    FLASH_ON_TIME, 0x06             ; 6 * 100 ms = 600 ms
    .equ    FLASH_OFF_TIME, 0x02            ; 2 * 100 ms = 200 ms
    .equ    ROLL_EFFECT_ANIMATION, 0x02     ; 2 * 100 ms = 200 ms

    ; IO port
    .equ    INPORT_ADDRESS, 0xFF00        ; input port address
    .equ    OUTPORT_ADDRESS, 0xFF00       ; output port address


;; ------------------------------------------------------------------------
------
    ;; startup section = PREPARE STARTUP SYSTEM
    .section .startup
    b   _start
    ldr pc, isr_addr
_start:
    ldr sp, tos_addr
    ldr pc, main_addr

tos_addr:
    .word   tos
main_addr:
    .word   main
isr_addr:
    .word   isr


;; ------------------------------------------------------------------------
------
    ;; .text = CODE
    .text
```

```
; void main()
; setup initial stuff, and handle the state machine
main:
    ; reset ticks
    mov r1, #0
    ldr r0, sysclk_addr2
    strb r1, [r0, #0]
    ; start ptc timer
    mov r0, #SYSCLK_FREQ
    bl  ptc_init
    ; setup cpsr
    mrs r0, cpsr
    mov r1, #CPSR_BIT_I
    orr r0, r0, r1
    msr cpsr, r0

    ; handle state machine (switch case)
switch_state:
    ldr r0, sm_state_addr
    ldrb r0, [r0, #0]       ; r0 = sm_state

case_state_wait_for_mode:
    mov r1, #STATE_WAIT_FOR_MODE
    cmp r0, r1
    bne case_state_get_bet
    bl wait_for_mode
    b case_state_end

case_state_get_bet:
    mov r1, #STATE_GET_BET
    cmp r0, r1
    bne case_state_wait_for_roll
    bl get_bet
    b case_state_end

case_state_wait_for_roll:
    mov r1, #STATE_WAIT_FOR_ROLL
    cmp r0, r1
    bne case_state_roll_effect
    bl wait_for_roll
    b case_state_end

case_state_roll_effect:
    mov r1, #STATE_ROLL_EFFECT
    cmp r0, r1
    bne case_state_roll_dice
    bl roll_effect
    b case_state_end
```

```
case_state_roll_dice:
    mov r1, #STATE_ROLL_DICE
    cmp r0, r1
    bne case_state_check_win
    bl roll_dice
    b case_state_end

case_state_check_win:
    mov r1, #STATE_CHECK_WIN
    cmp r0, r1
    bne case_state_final_timer
    bl check_win
    b case_state_end

case_state_final_timer:
    mov r1, #STATE_FINAL_TIMER
    cmp r0, r1
    bne case_state_default
    bl final_timer
    b case_state_end

case_state_default:
    mov r0, #STATE_WAIT_FOR_MODE
case_state_end:
    b switch_state

sysclk_addr2:
    .word sysclk


; void change_state(uint8_t new_state)
; changes sm_state variable to [new_state]
change_state:
    ldr r1, sm_state_addr
    strb r0, [r1, #0]
    mov pc, lr

sm_state_addr:
    .word sm_state


; void show_hex_sevseg(uint8_t value, bool dot_mask)
; shows display_numbers_array[value] on sevseg,
show_hex_sevseg:
    push lr
    ldr r2, display_numbers_array_addr
    ldrb r2, [r2, r0]
    mov r0, r2
```

```
    orr r0, r0, r1
    bl outport_write
    pop pc

display_numbers_array_addr:
    .word display_numbers_array


; void wait_for_mode() (STATE 0)
; turns off sevseg and waits for I0 to start
wait_for_mode:
    push lr
    ldr r1, win_flag_addr2
    mov r0, #0
    strb r0, [r1, #0]   ; reset win to false
    mov r0, #SEV_SEG_OFF ; clear output port
    bl outport_write
wait_for_mode_loop:
    bl get_mode
    mov r1, #1
    cmp r0, r1  ; if mode != 1
    bne wait_for_mode_loop
wait_for_mode_exit:
    mov r0, #STATE_GET_BET
    bl change_state
    pop pc

win_flag_addr2:
    .word win_flag


; void get_bet() (STATE 2)
; displays current bet value (I2..5) and if MODE is 0, waits for roll
get_bet:
    push lr
get_bet_loop:
    bl get_bet_value ; r0 = bet value
    ldr r1, bet_value_addr
    strb r0, [r1, #0] ; BET = get_bet_value()
    mov r1, #DOT_ON ; dot on
    bl show_hex_sevseg ; display value (r0) on sevseg
    bl get_mode
    mov r1, #0
    cmp r0, r1  ; if mode != 0
    bne get_bet_loop
get_bet_exit:
    mov r0, #STATE_WAIT_FOR_ROLL
    bl change_state
    pop pc
```

```
bet_value_addr:
    .word bet_value


; void wait_for_roll() (STATE 3)
; turns off sevseg and waits for ascending transition of I7 to start
wait_for_roll:
    push lr
    push r4
    ldr r1, bet_value_addr3
    ldrb r0, [r1, #0] ; BET = get_bet_value()
    mov r1, #0
    bl show_hex_sevseg ; display value (r0) on sevseg
wait_for_roll_loop:
    bl get_roll
    mov r1, #1
    cmp r0, r1
    bne wait_for_roll_loop
wait_for_0:
    bl get_roll
    mov r1, #0
    cmp r0, r1
    bne wait_for_0
wait_for_roll_exit:
    mov r0, #STATE_ROLL_EFFECT
    bl change_state
    pop r4
    pop pc

bet_value_addr3:
    .word bet_value

; void roll_effect() (STATE 4)
; light effect from a. -> f. de 200ms
roll_effect:
    push lr
    push r4 ; i (5 times counter)
    push r5 ; j (animation counter)
    push r6 ; current tick
    mov r4, #0  ; i = 0
    mov r5, #0 ; j = 0
roll_effect_loop: ; five time loop
roll_effect_animation_loop: ; animation loop
    bl sysclk_get_ticks
    mov r6, r0                          ; r6 = current time ticks
    ldr r0, display_roulette_array_addr ; r0 = &display_roulette_array
    ldrb r0, [r0, r5]                   ; r0 = display_roulette_array[r5]
    bl outport_write
```

```
    mov r2, #ANIMATION_SIZE
    cmp r5, r2          ; if j > 6
    bhs roll_effect_loop_iterate
    add r5, r5, #1      ; j++
roll_effect_animation_sleep:    ; wait 200 ms between each animation
    mov r0, r6  ; r0 = r6 (sysclk_elapsed parameter)
    bl sysclk_elapsed
    mov r1, #ROLL_EFFECT_ANIMATION
    cmp r0, r1                  ; if elapsed < time, continue looping
    blo roll_effect_animation_sleep
    b roll_effect_animation_loop
roll_effect_loop_iterate:
    mov r0, #ROLL_EFFECT_COUNT
    cmp r4, r0      ; if i > 5
    bhs roll_effect_loop_end
    add r4, r4, #1  ; i++
    mov r5, #0 ; reset j
    b roll_effect_loop
roll_effect_loop_end:
    mov r0, #STATE_ROLL_DICE
    bl change_state
    pop r6
    pop r5
    pop r4
    pop pc

display_roulette_array_addr:
    .word display_roulette_array


; void roll_dice() (STATE 5)
; generate random number and show it
roll_dice:
    push lr
    bl rand
    mov r1, #RAND_MASK
    and r0, r0, r1
    ldr r1, random_dice_value_addr
    strb r0, [r1, #0] ; RANDOM_DICE = get_random_value()
    mov r1, #0
    bl show_hex_sevseg  ; show random value in sevseg
    mov r0, #STATE_CHECK_WIN
    bl change_state
    pop pc


; void check_win() (STATE 6)
; checks if player won (BET == RANDOM_DICE_VALUE)
check_win:
```

```
    push lr
    ldr r0, bet_value_addr2
    ldrb r0, [r0, #0]
    ldr r1, random_dice_value_addr
    ldrb r1, [r1, #0]
    cmp r0, r1 ; if (BET != RANDOM_DICE_VALUE)
    bne check_win_exit
    ldr r1, win_flag_addr
    mov r0, #1
    strb r0, [r1, #0]   ; win = true
check_win_exit:
    mov r0, #STATE_FINAL_TIMER
    bl change_state
    pop pc

random_dice_value_addr:
    .word random_dice_value


; void final_timer() (STATE 7)
; check win and waits 10 secs
; if it's a win the sevseg blinks with a rhythm of 800ms and a duty cicle
of 75% (600ms on and 200ms off)
final_timer:
    push lr
    push r4
    bl sysclk_get_ticks
    mov r4, r0
final_timer_loop:
    ldr r0, win_flag_addr
    ldrb r0, [r0, #0]
    mov r1, #1
    cmp r0, r1
    bne check_timer_exit
flash_win:
    bl flash_on
    bl flash_off
check_timer_exit:
    mov r0, r4
    bl sysclk_elapsed
    mov r1, #TEN_SECS
    cmp r0, r1
    bge final_timer_exit
    b final_timer_loop

final_timer_exit:
    mov r0, #STATE_WAIT_FOR_MODE
    bl change_state
    pop r4
```

```
    pop pc

bet_value_addr2:
    .word bet_value

win_flag_addr:
    .word win_flag


; bool get_mode()
; returns MODE value (true = 1, false = 0) (I0)
get_mode:
    push lr
    bl inport_read
    mov r1, #MODE_MASK
    and r0, r0, r1
    pop pc


; uint8_t get_bet_value()
; returns bet value (4 bits)
get_bet_value:
    push lr
    bl inport_read
    mov r1, #BET_MASK
    and r0, r0, r1
    lsr r0, r0, #2 ; value = (inport_read() & 0b00111100) >> 2
    pop pc


; bool get_roll()
; returns roll value  (true = 1, false = 0) (I7)
get_roll:
    push lr
    bl inport_read
    mov r1, #ROLL_MASK
    and r0, r0, r1
    lsr r0, r0, #7 ; value = (inport_read() & 0b00111100) >> 7
    pop pc


; void flash_on()
; turnes the sevseg on (with random_dice_value) and sleeps for
FLASH_ON_TIME
flash_on:
    push lr
    push r4 ; r4 will store the current tick when joining this routine
    ldr r0, random_dice_value_addr2
    ldrb r0, [r0, #0]   ; r0 = random_dice_value
```

```
    mov r1, #0
    bl show_hex_sevseg  ; display random_dice_value
    bl sysclk_get_ticks
    mov r4, r0
flash_on_loop:
    mov r0, r4  ; r0 = r4
    bl sysclk_elapsed
    mov r1, #FLASH_ON_TIME
    cmp r0, r1  ; if elapsed(r4) < FLASH_ON_TIME keep looping
    blo flash_on_loop
    pop r4
    pop pc


; void flash_off()
; turnes the flash of and sleeps for FLASH_OFF_TIME
flash_off:
    push lr
    push r4 ; r4 will store the current tick when joining this routine
    mov r0, #SEV_SEG_OFF
    bl outport_write    ; turn off sevseg
    bl sysclk_get_ticks
    mov r4, r0
flash_off_loop:
    mov r0, r4
    bl sysclk_elapsed
    mov r1, #FLASH_OFF_TIME
    cmp r0, r1  ; if elapsed(r4) < FLASH_OFF_TIME keep looping
    blo flash_off_loop
    pop r4
    pop pc

random_dice_value_addr2:
    .word random_dice_value


; uint32_t umull32(uint32_t M, uint32_t m)
; r1:r0 = m
; r3:r2 = M
; r6 = p_1
; r7 = i
; r8 = tmp
; return value stored in r0:r1
umull32:
    ; r5:r4:r1:r0 = p (aproveitar memoria porque nao vamos chamar outras
rotinas)
    push r4
    push r5
    push r6 ; r6 = p_1
```

```
    push r7 ; r7 = i
    push r8 ; r8 = tmp
    ; int64_t p = m;
    mov r4, #0  ; p[16..23] = 0
    mov r5, #0  ; p[24..31] = 0
    mov r6, #0  ; uint8_t p_1 = 0;
umull32_for:
    mov r7, #0  ; i = 0
umull32_loop:
umull32_if:
    mov r8, #1
    and r8, r0, r8  ; (p & 0x1)
    bzc umull32_else_if
    mov r8, #1
    cmp r6, r8      ; p_1 == 1
    bne umull32_else_if
    add r4, r4, r2  ;   p += M << 32
    adc r5, r5, r3
umull32_else_if:
    mov r8, #0x1
    and r8, r0, r8  ; (p & 0x1)
    bzs umull32_else
    mov r8, #0
    cmp r6, r8      ; p_1 == 0
    bne umull32_else
    sub r4, r4, r2 ;    p -= M << 32
    sbc r5, r5, r3
umull32_else:
    mov r8, #1
    and r8, r0, r8  ; r8 = p & 0x1
    mov r6, r8      ; p_1 = r8 = p & 0x1;
    asr r5, r5, #1  ; p = p >> 1
    rrx r4, r4
    rrx r1, r1
    rrx r0, r0
    add r7, r7, #1 ; i++
    mov r8, #32
    cmp r7, r8
    blo umull32_loop ; if i < 32, loop again
umull32_for_end:
    ; return value is already stored in r1:r0
    pop r8
    pop r7
    pop r6
    pop r5
    pop r4
    mov pc, lr
```

```
; void srand (uint32_t nseed)
; nseed = r1:r0
; r2 = &seed
srand:
    ldr r2, seed_addr   ; r2 = &seed
    str r0, [r2, #0]    ; seed[0..15] = r0
    str r1, [r2, #2]    ; seed[16..31] = r1
    mov pc, lr


; uint16_t rand()
; r0/r1/r4/r5/r6 = tmp
; r2 = seed[0..15]
; r3 = seed[16..31]
; return value stored in r0
rand:
    push lr
    push r4
    ; prepare arguments for umull32(seed(r3:r2), 214013(r1:r0))
    ; seed = r3:r2
    ldr r0, seed_addr
    ldr r2, [r0, #0]
    ldr r3, [r0, #2]
    ; 214013 = 0x343FD = r1:r0
    mov r0, 0xfd
    movt r0, 0x43
    mov r1, 0x03
    movt r1, 0x00
    ; r1:r0 = umull32(seed, 214013)
    bl umull32
    ; add r1:r0 (32bits) with 2531011 = 0x269EC3 (24 bits)
    mov r4, 0xc3
    movt r4, 0x9e
    add r0, r0, r4
    mov r4, 0x26
    movt r4, 0x00
    adc r1, r1, r4
    bl get_mod      ;modulo function which returns in (r1:r0):
    ldr r2, seed_addr
    ; seed = ... % RAND_MAX
    str r0, [r2, #0]
    str r1, [r2, #2]
    ; return seed >> 16
    mov r0, r1 ; seed >> 16
    mov r1, #0
    pop r4
    pop pc

seed_addr:
```

```
    .word seed


; uint32_t get_remainder(uint32_t num)
; num = r1:r0
; RAND_MAX = r2:r2
; tmp = r3
; return value stored in r1:r0
get_mod:
    mov r2, #RAND_MAX_L
    movt r2, #RAND_MAX_H
get_mod_check:
    cmp r1, r2
    bne mod_not_zero
    cmp r0, r2
    bne mod_not_zero
    ; store return value
    mov r0, #0
    mov r1, #0
    b mod_return
mod_not_zero:
mod_return:
    mov pc, lr


; sysclk_elapsed elapsed(uint16_t t0)
; returns (sysclk_get_tick() - t0)
sysclk_elapsed:
    push lr
    push r0
    bl sysclk_get_ticks
    pop r1
    sub r0, r0, r1
    pop pc


; uint16_t sysclk_get_ticks();
; returns: current sysclk global variable value
sysclk_get_ticks:
    ldr r1, sysclk_addr
    ldr r0, [r1, #0]
    mov pc, lr

sysclk_addr:
    .word   sysclk


; void isr()
; interruption function, increment sysclk
```

```
isr:
    push lr
    push r0
    push r1
    push r2
    push r3
    bl sysclk_get_ticks
    add r0, r0, #1
    str r0, [r1, #0]
    mov r0, #1
    ldr r1, PTC_ADDR
    strb r0, [ r1, PTC_TIR ]
    pop r3
    pop r2
    pop r1
    pop r0
    pop lr
    movs pc, lr


; uint8_t inport_read()
; reads and returns value from INPUT PORT.
inport_read:
    ldr r1, inport_addr
    ldrb r0, [r1, #0]
    mov pc, lr

inport_addr:
    .word   INPORT_ADDRESS


; void outport_write(uint16_t value)
; writes [value] aKa r0 to OUTPUT PORT
outport_write:
    ldr r1, outport_addr
    strb r0, [r1, #0]
    mov pc, lr


; void outport_clear_bits( uint8_t pins_mask )
; clears all bits masked from OUTPUT PORT
outport_clear_bits:
    push lr
    ldr r1, outport_img_addr
    ldrb r2, [r1, #0]
    mvn r0, r0
    and r0, r2, r0
    strb r0, [r1]
    bl  outport_write
```

```
    pop pc


; void outport_init(uint8_t value)
; inits the output port with [value].
outport_init:
    push lr
    ldr r1, outport_img_addr
    strb r0, [r1]
    bl  outport_write
    pop pc

outport_addr:
    .word   OUTPORT_ADDRESS

outport_img_addr:
    .word   outport_img


; void ptc_start()
; starts the pTC counter
ptc_start:
    ldr r0, PTC_ADDR
    mov r1, #PTC_CMD_START
    strb    r1, [r0, #PTC_TCR]
    mov pc, lr


; void ptc_stop()
; stops the pTC counter, setting TC register to 0
ptc_stop:
    ldr r0, PTC_ADDR
    mov r1, #PTC_CMD_STOP
    strb    r1, [r0, #PTC_TCR]
    mov pc, lr


; uint8_t ptc_get_value()
; returns actual value of pTC counter
ptc_get_value:
    ldr r1, PTC_ADDR
    ldrb r0, [r1, #PTC_TC]
    mov pc, lr


; void ptc_init( uint8_t interval )
; inits a new pTC counter with [interval] count (ticks)
; it also cleans interrupt pending requests if there are any
ptc_init:
```

```
    push lr
    push r4
    mov r4, r0
    bl ptc_stop
    ldr r1, PTC_ADDR
    strb r4, [r1, #PTC_TMR] ; meter TMR a interval
    strb r1, [r1, #PTC_TIR] ; limpar PTC TIR
    bl ptc_start
    pop r4
    pop pc

PTC_ADDR:
    .word   PTC_ADDRESS


;; -----------------------------------------------------------------------
------
    ;; .data = INITIALIZED GLOBAL DATA
    .data

seed:
    .word 0x0001
    .word 0x0000

display_numbers_array:
    .byte 0x3F, 0x6, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x7, 0x7F, 0x6F, 0x77,
0x7C, 0x39, 0x5e, 0x79, 0x71

display_roulette_array:
    .byte 0x1, 0x20, 0x10, 0x8, 0x4, 0x2, 0x1

sm_state: ; state machine state
    .byte 0

random_dice_value:
    .byte 0

bet_value:
    .byte 0

win_flag:
    .byte 0


;; -----------------------------------------------------------------------
------
    ;; .bss = NON INITIALIZED GLOBAL DATA
    .section .bss
```

```
outport_img:
    .space  1
    .align

sysclk:
    .space  2


;; ------------------------------------------------------------------------
------
    ;; .section .stack = IMPLEMENT STACK
    .section .stack
    .space  STACK_SIZE
tos:
```

5. Conclusão

Concluindo, com este trabalho foi possível aprofundar as nossas *skills* de desenvolvimento de programas, já com alguma extensão em linguagem *assembly*. Para além disso, aplicar uma correta utilização do circuito pTC de modo a, neste caso, obtermos intervalos de tempo que é algo bastante útil. É também importante referir que os objetivos deste trabalho foram alcançados, visto que o programa foi testado e validado pelo docente sem a descoberta de quaisquer erros ou bugs.