

ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
DISCIPLINA: LABORATÓRIO DE PROCESSADORES- PCS3732
1º QUADRIMESTRE/2021



Aula 10
15 de Julho de 2021

GRUPO 10

Arthur Pires da Fonseca
Bruno José Mório
Iago Soriano Roque Monteiro

NUSP: 10773096
NUSP: 10336852
NUSP: 8572921

Sumário

10.3 Geração do código e como se roda.	3
10.5 Corrija os ERROS na apostila	3
10.5.1 O código da apostila não inicializa adequadamente as pilhas na placa versatile	3
10.5.2 O retorno de IRQ é feito de forma ERRADA	3
10.5.3 Problema ao recuperar o cpsr anterior	3
10.5.4 Polemica - BNE x BLNE	3
10.6 Observe e faça pequenas alterações no código em assembly	6
10.6.1 Examine no debugger:	6
10.6.2 Desabilite as instruções	11
10.6.3 Endereço do vetor de interrupcao	12
10.6.4 Timer, inicialização	13
10.6.5 Fazendo o tratamento da interrupção em C.	14
Apêndice	15
10.6.5	15
handler.c	15
handler.s	15
irq.s	16

10.3 Geração do código e como se roda.

Utilizamos o comando abaixo para compilar os arquivos startup.s, que contém o vetor de interrupções e os handlers, script.c, que contém as funções que imprimem “#” e ““, e rodar o script de link, produzir o binário e rodá-lo.

```
eabi-gcc script.c -o script.o && eabi-as startup.s -o startup.o && eabi-ld -T irqld.ld startup.o script.o -o irq.elf && eabi-bin irq.elf irq.bin && qemu irq.bin
```

10.5 Corrija os ERROS na apostila

10.5.1 O código da apostila não inicializa adequadamente as pilhas na placa versatile

10.5.2 O retorno de IRQ é feito de forma ERRADA

10.5.3 Problema ao recuperar o cpsr anterior

10.5.4 Polemica - BNE x BLNE

Segue abaixo o nosso código final em startup.s.

Estão destacadas as correções feitas em cada um dos itens acima.

```
.global _start
.text

_start:
    b _Reset @posição 0x00 - Reset
    ldr pc, _undefined_instruction @posição 0x04 - Instrução não-definida
    ldr pc, _software_interrupt @posição 0x08 - Interrupção de Software
    ldr pc, _prefetch_abort @posição 0x0C - Prefetch Abort
    ldr pc, _data_abort @posição 0x10 - Data Abort
    ldr pc, _not_used @posição 0x14 - Não utilizado
    ldr pc, _irq @posição 0x18 - Interrupção (IRQ)
    ldr pc, _fiq @posição 0x1C - Interrupção(FIQ)

_undefined_instruction: .word undefined_instruction
_software_interrupt: .word software_interrupt
_prefetch_abort: .word prefetch_abort
_data_abort: .word data_abort
_not_used: .word not_used

_irq: .word irq
_fiq: .word fiq

INTPND: .word 0x10140000 @Interrupt status register
```

```

INTSEL: .word 0x1014000C @interrupt select register( 0 = irq, 1 = fiq)
INTEN: .word 0x10140010 @interrupt enable register
TIMER0L: .word 0x101E2000 @Timer 0 load register
TIMER0V: .word 0x101E2004 @Timer 0 value registers
TIMER0C: .word 0x101E2008 @timer 0 control register
TIMER0X: .word 0x101E200c @timer 0 interrupt clear register

```

```
_Reset:
```

```
10.5.1 /*
```

```

    MRS r0, cpsr                @ salvando o modo corrente em R0
    MSR cpsr_ctl, #0b11010010   @ alterando para modo interrupt
    LDR sp, =timer_stack_top     @ a pilha de interrupções de tempo é setada
    MSR cpsr, r0                 @ retorna para o modo anterior
*/

```

```
    LDR sp, =stack_top
```

```
    bl main
```

```
    b .
```

```
undefined_instruction:
```

```
    b .
```

```
software_interrupt:
```

```
    b do_software_interrupt @vai para o handler de interrupções de software
```

```
prefetch_abort:
```

```
    b .
```

```
data_abort:
```

```
    b .
```

```
not_used:
```

```
    b .
```

```
irq:
```

```
    b do_irq_interrupt @vai para o handler de interrupções IRQ
```

```
fiq:
```

```
    b .
```

```
do_software_interrupt: @Rotina de Interrupção de software
```

```
    add r1, r2, r3 @r1 = r2 + r3
```

```
    mov pc, r14 @volta p/ o endereço armazenado em r14
```

```
do_irq_interrupt: @Rotina de interrupções IRQ
```

```

STMFD sp!, {r0 - r3, LR} @Empilha os registradores

LDR r0, INTEND @Carrega o registrador de status de interrupção
LDR r0, [r0]
TST r0, #0x0010 @verifica se é uma interrupção de timer

10.5.3 parte 1 /*
    STMFD sp!, {pc} @ salva pc na pilha do modo de interrupções
*/

10.5.4 /*
    BLNE handler_timer @vai para o rotina de tratamento da interrupção de timer
*/
    LDMFD sp!, {r0 - r3,lr} @retorna

10.5.2 /*
    sub lr, lr, #4 @ corrigindo o lr
*/

    STMFD sp!, {lr}
10.5.3 parte 3 /*
    LDMFD sp!, {pc}^
*/
handler_timer:
10.5.3 parte 2 /*
    STMFD sp!,{R0-R12}
*/
    LDR r0, TIMER0X
    MOV r1, #0x0
    STR r1, [r0] @Escreve no registrador TIMER0X para limpar o pedido de
    interrupção

    BL print_interrupcao

    LDMFD sp!,{R0-R12}
    LDMFD sp!, {pc}

    mov pc, r14 @retorna

timer_init:
    LDR r0, INTEN
    LDR r1,=0x10 @bit 4 for timer 0 interrupt enable
    STR r1,[r0]

    LDR r0, TIMER0C
    LDR r1, [r0]
    MOV r1, #0xA0 @enable timer module
    STR r1, [r0]

```

```

LDR r0, TIMER0V
MOV r1, #0xff @setting timer value
STR r1,[r0]

mrs r0, cpsr
bic r0,r0,#0x80
msr cpsr_c,r0 @enabling interrupts in the cpsr

mov pc, lr

main:
    bl timer_init @initialize interrupts and timer 0
stop:
    BL print_loop
    b stop

```

10.6 Observe e faça pequenas alterações no código em assembly

10.6.1 Examine no debugger:

Debugue o código: rode até a instrução `LDMFD sp!,{R0-R12,pc}^` e examine a pilha. Verifique se o valor a ser carregado em PC está correto.

The screenshot shows a debugger window with two panes. The left pane displays the assembly code for a program, and the right pane shows the current state of the registers and the stack.

Assembly Code (Left Pane):

```

84 BLNE handler_timer @vai para o rotina de tratamento da interrupção d
85 LDMFD sp!, {r0 - r3,lr} @retorna
86
87 sub lr, lr, #4 @ corrigindo o lr
88 STMFD sp!, {lr}
89 LDMFD sp!, {pc}^
90 @mov pc, r14^
91
92 handler_timer:
93 STMFD sp!,{R0-R12}
94
95 LDR r0, TIMER0X
96 MOV r1, #0x0

```

Register Group: general (Right Pane):

Register	Value	Address
r0	0x1b8	440
r1	0xff	255
r2	0x20	32
r3	0x0	0
r4	0x0	0
r5	0x0	0
r6	0x0	0
r7	0x0	0
r8	0x0	0
r9	0x0	0
r10	0x0	0
r11	0x1b0	4528
r12	0x0	0
sp	0x21b8	0x21b8
pc	0xbc	0xbc <do_irq_interru
lr	0x170	368
cpsr	0x60001d2	1610613202

Debugger Output (Bottom Pane):

```

remote Thread 1 In: do_irq_interrupt Line: 89 PC: 0xbc
Transfer rate: 3552 bits in <1 sec, 222 bytes/write.
(gdb) b 89
Breakpoint 1 at 0xbc: file startup.s, line 89.
(gdb) c
Continuing.

Breakpoint 1, do_irq_interrupt () at startup.s:89
(gdb) si
print_uart0 (s=0x1b9 "") at script.c:8
(gdb) c
Continuing.

Breakpoint 1, do_irq_interrupt () at startup.s:89
(gdb)

```

Saída do programa (esquerda) e parada no código (direita) antes da volta para o modo Supervisor (últimos 5 bits = 0x13). Caractere “#” já printado

```

Register group: general
r0 0x1b8 440 r1 0xff 255
r2 0x20 32 r3 0x0 0
r4 0x0 0 r5 0x0 0
r6 0x0 0 r7 0x0 0
r8 0x0 0 r9 0x0 0
r10 0x0 0 r11 0x11b0 4528
r12 0x0 0 r13 0x21b8 0x21b8
r14 0x0 0 r15 0x0 0
r16 0x170 368 r17 0x0 0
r18 0x0 0 r19 0x0 0
r20 0x0 0 r21 0x0 0
r22 0x0 0 r23 0x0 0
r24 0x0 0 r25 0x0 0
r26 0x0 0 r27 0x0 0
r28 0x0 0 r29 0x0 0
r30 0x0 0 r31 0x0 0
cpsr 0x600001d2 1610613202

--startup.s--
84 BLNE handler_timer @vai para o rotina de tratamento da interrupção d
85 LDMFD sp!, {r0 - r3,lr} @retorna
86
87 sub lr, lr, #4 @ corrigindo o lr
88 STMFD sp!, {lr}
B> 89 LDMFD sp!, {pc}
90 @mov pc, r14^
91
92 handler_timer:
93 STMFD sp!, {R0-R12}
94
95 LDR r0, TIMER0X
96 MOV r1, #0x0

remote Thread 1 In: do_irq_interrupt Line: 89 PC: 0x170
Continuing.

Breakpoint 1, do_irq_interrupt () at startup.s:89
(gdb) si
print_uart0 (s=0x1b9 "") at script.c:8
(gdb) c
Continuing.

Breakpoint 1, do_irq_interrupt () at startup.s:89
(gdb) x/10 $sp
0x21b8: 368 0 0 0
0x21c8: 0 0 0 0
0x21d8: 0 0 0 0
(gdb)

```

Valores da pilha antes da volta para o modo Supervisor (valor de LR para ser devolvido para o PC)

```

Register group: general
r0 0x1b8 440 r1 0xff 255
r2 0x20 32 r3 0x0 0
r4 0x0 0 r5 0x0 0
r6 0x0 0 r7 0x0 0
r8 0x0 0 r9 0x0 0
r10 0x0 0 r11 0x11b0 4528
r12 0x0 0 r13 0x21b8 0x21b8
r14 0x0 0 r15 0x0 0
r16 0x170 368 r17 0x0 0
r18 0x0 0 r19 0x0 0
r20 0x0 0 r21 0x0 0
r22 0x0 0 r23 0x0 0
r24 0x0 0 r25 0x0 0
r26 0x0 0 r27 0x0 0
r28 0x0 0 r29 0x0 0
r30 0x0 0 r31 0x0 0
cpsr 0x60000153 1610613075

--script.c--
3 void print_uart0(const char *s) {
4     while(*s != '\0') { /* Loop until end of string */
5         *UART0DR = (unsigned int)(*s); /* Transmit char */
6         s++; /* Next char */
7     }
8 }
9
10 void print_interrupcao()
11 {
12     print_uart0("#");
13 }
14
15 void print_loop()

remote Thread 1 In: print_uart0 Line: 8 PC: 0x170
Breakpoint 1, do_irq_interrupt () at startup.s:89
(gdb) si
print_uart0 (s=0x1b9 "") at script.c:8
(gdb) c
Continuing.

Breakpoint 1, do_irq_interrupt () at startup.s:89
(gdb) x/10 $sp
0x21b8: 368 0 0 0
0x21c8: 0 0 0 0
0x21d8: 0 0 0 0
(gdb) si
print_uart0 (s=0x1b9 "") at script.c:8
(gdb)

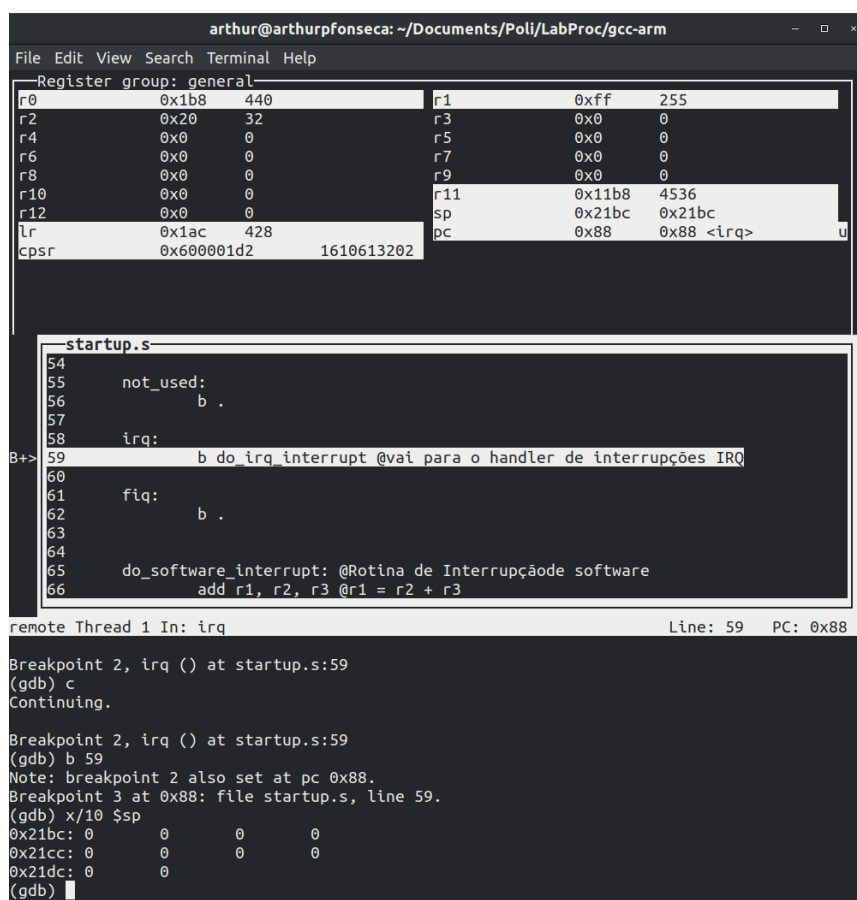
```

Volta para o modo Supervisor, onde havia parado antes da Interrupção de Hardware

Coloque um breakpoint logo no início da interrupção de timer, o mais perto possível do vetor de interrupção, e rode até lá. Qual é o modo em que o processador roda? Qual é o sp? O que se vê na pilha? Relacione o PC que deveria apontar para a instrução seguinte antes de ocorrer a interrupção com o LR (veja que tem uma diferença de 4 bytes).

O processador está no modo de Interrupção (últimos 5 bits do CPSR = “0x12”) no momento em que a interrupção de hardware acontece.

Conforme demonstra a imagem abaixo, \$sp = 0x21bc, a pilha contém apenas zeros (espaço aberto pelo linker)



```
arthur@arthurpfonseca: ~/Documents/Pol/LabProc/gcc-arm
File Edit View Search Terminal Help
Register group: general
r0 0x1b8 440 r1 0xff 255
r2 0x20 32 r3 0x0 0
r4 0x0 0 r5 0x0 0
r6 0x0 0 r7 0x0 0
r8 0x0 0 r9 0x0 0
r10 0x0 0 r11 0x11b8 4536
r12 0x0 0 sp 0x21bc 0x21bc
lr 0x1ac 428 pc 0x88 0x88 <irq>
cpsr 0x600001d2 1610613202

startup.s
54
55 not_used:
56 b .
57
58 irq:
59 b do_irq_interrupt @vai para o handler de interrupções IRQ
60
61 fiq:
62 b .
63
64
65 do_software_interrupt: @Rotina de Interrupção de software
66 add r1, r2, r3 @r1 = r2 + r3

remote Thread 1 In: irq Line: 59 PC: 0x88

Breakpoint 2, irq () at startup.s:59
(gdb) c
Continuing.

Breakpoint 2, irq () at startup.s:59
(gdb) b 59
Note: breakpoint 2 also set at pc 0x88.
Breakpoint 3 at 0x88: file startup.s, line 59.
(gdb) x/10 $sp
0x21bc: 0 0 0
0x21cc: 0 0 0
0x21dc: 0 0
(gdb)
```

Código parado no vetor de interrupção, logo antes da chamada da rotina de tratamento da interrupção

Após o tratamento da interrupção, pode-se ver a diferença de 4 bytes entre o LR original (0x1ac) e o final (0x1a8). Vide próximas duas imagens.


```

arthur@arthurfonseca: ~/Documents/Poli/LabProc/gcc-arm
File Edit View Search Terminal Help
Register group: general
r0      0x1b8  440    r1      0xff   255
r2      0x20   32    r3      0x0    0
r4      0x0    0     r5      0x0    0
r6      0x0    0     r7      0x0    0
r8      0x0    0     r9      0x0    0
r10     0x0    0    r11     0x11b8 4536
r12     0x0    0    sp      0x21b8 0x21b8
lr      0x1a8  424    pc      0xbc   0xbc <do_irq_interrupt>
cpsr    0x600001d2 1610613202

startup.s
80
81      STMFD sp!, {pc}          @ salva pc na pilha do modo de interr
82      @MSR cpsr, r0           @ retorna para o modo anterio
83
84      BLNE handler_timer @vai para o rotina de tratamento da interrupção d
85      LDMFD sp!, {r0 - r3,lr} @retorna
86
87      sub lr, lr, #4 @ corrigindo o lr
88      STMFD sp!, {lr}
> 89      LDMFD sp!, {pc}^
90      @mov pc, r14^
91
92      handler_timer:

remote Thread 1 In: do_irq_interrupt Line: 89 PC: 0xbc
(gdb) si
do_irq_interrupt () at startup.s:70
do_irq_interrupt () at startup.s:84
handler_timer () at startup.s:93
(gdb) n
handler_timer () at startup.s:103
do_irq_interrupt () at startup.s:85
do_irq_interrupt () at startup.s:87
do_irq_interrupt () at startup.s:89
(gdb) x/10 $sp
0x21b8: 424 0 0 0
0x21c8: 0 0 0 0
0x21d8: 0 0 0 0
(gdb)

```

Valor de LR subtraído de 4 na pilha, para ser passado para o PC

```

arthur@arthurfonseca: ~/Documents/Poli/LabProc/gcc-arm
File Edit View Search Terminal Help
Register group: general
r0      0x1b8  440    r1      0xff   255
r2      0x20   32    r3      0x0    0
r4      0x0    0     r5      0x0    0
r6      0x0    0     r7      0x0    0
r8      0x0    0     r9      0x0    0
r10     0x0    0    r11     0x11b8 4536
r12     0x0    0    sp      0x11b4 0x11b4
lr      0x1a8  424    pc      0x1a8 0x1a8 <print_loop+16>
cpsr    0x60000153 1610613075

script.c
13      }
14
15      void print_loop()
16      {
17          print_uart0(" ");
> 18      }^?
19
20
21
22
23
24
25

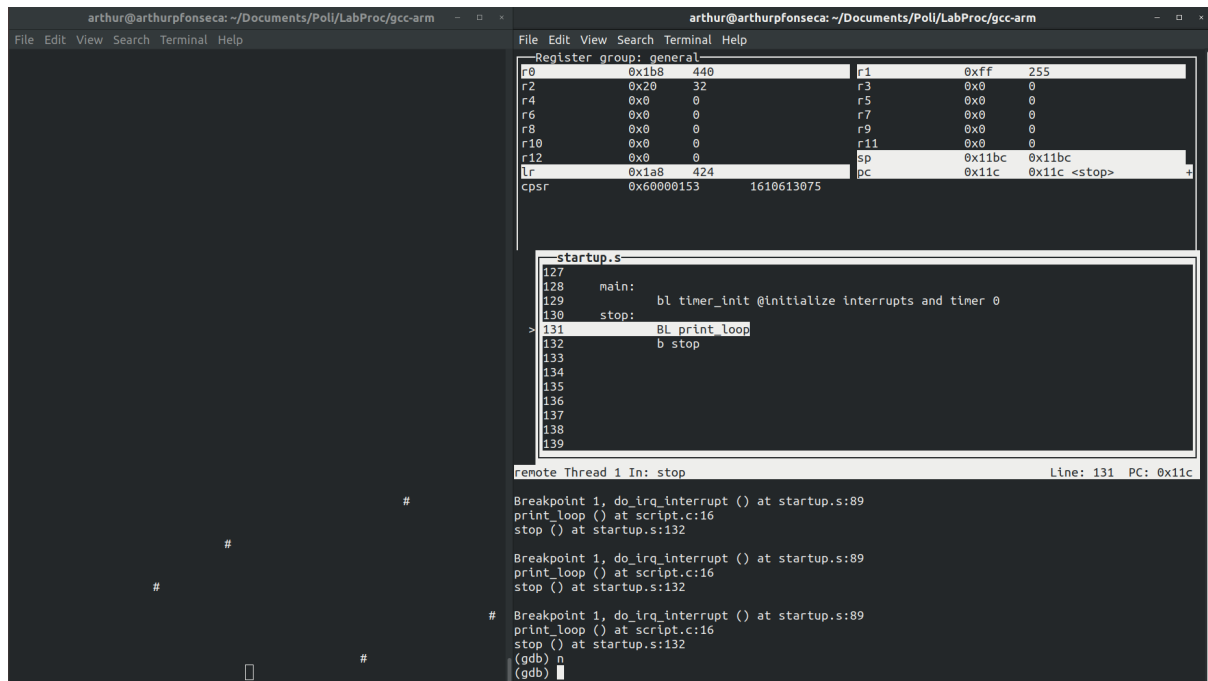
remote Thread 1 In: print loop Line: 18 PC: 0x1a8
do_irq_interrupt () at startup.s:84
handler_timer () at startup.s:93
(gdb) n
handler_timer () at startup.s:103
do_irq_interrupt () at startup.s:85
do_irq_interrupt () at startup.s:87
do_irq_interrupt () at startup.s:89
(gdb) x/10 $sp
0x21b8: 424 0 0 0
0x21c8: 0 0 0 0
0x21d8: 0 0 0 0
(gdb) si
print_loop () at script.c:18
(gdb)

```

Volta ao ponto no código que seria rodado antes da interrupção ter sido feita

Como o processador chaveia de modo ao sair da rotina de interrupção? Verifique isso localizando a instrução que faz isso. Quando estamos debugando passo a passo e observamos que o processador sai da rotina de interrupção, pode acontecer do timer da máquina virtual continuar marcando o tempo e como nós seres humanos somos muito mais lentos que a máquina, é bem provável que logo ao sair da rotina de interrupção, já tenha ocorrido outra interrupção de timer. Verifique se isso acontece.

A instrução que chaveia o processador de volta para o modo Supervisor é *LDMFD sp!, {pc}^*.



The screenshot shows the GDB interface with two main panes. The top pane displays the 'Register group: general' with values for registers r0 through r12, lr, cpsr, sp, and pc. The bottom pane shows the source code for 'startup.s' with line numbers 127 to 139. The current instruction is at line 131: 'BL print_loop'. The status bar at the bottom indicates 'remote Thread 1 In: stop' and 'Line: 131 PC: 0x11c'.

Register	Value
r0	0x1b8 440
r2	0x20 32
r4	0x0 0
r6	0x0 0
r8	0x0 0
r10	0x0 0
r12	0x0 0
lr	0x1a8 424
cpsr	0x60000153 1610613075
sp	0x11bc 0x11bc
pc	0x11c 0x11c <stop>

```
127
128 main:
129     bl timer_init @Initialize interrupts and timer 0
130 stop:
131     BL print_loop
132     b stop
133
134
135
136
137
138
139
```

Saída do programa (esquerda) e visualização do código no GDB (direita) após sucessivos comandos de “next”

Verificamos que, ao debugar e rodar a instrução de print “ “ passo a passo, há diferença no número de “ “ que são impressos entre cada “#”.

Isso mostra que o timer continua rodando enquanto debugamos o código.

10.6.2 Desabilite as instruções

Qual é a instrução que habilita as interrupções? Desabilite as interrupções no cpsr e rode. O que acontece? Você acha razoável habilitar as interrupções enquanto se programa o timer? De fato isso não é razoável, primeiro deve-se programar o timer para depois habilitar as interrupções. Isto é: alterar cpsr zerando o bit I deve ser a última operação a ser feita. Altere isso no código. Veja se funciona.

```
timer_init:
    LDR r0, INTEN
    LDR r1,=0x10 @bit 4 for timer 0 interrupt enable
    STR r1,[r0]

    LDR r0, TIMER0C
    LDR r1, [r0]
    MOV r1, #0xA0 @enable timer module
    STR r1, [r0]

    LDR r0, TIMER0V
    MOV r1, #0xff @setting timer value
    STR r1,[r0]

    mrs r0, cpsr
    bic r0,r0,#0x80
    msr cpsr_c,r0 @enabling interrupts in the cpsr

    mov pc, lr
```

As instruções que habilitam as interrupções são as seguintes:

```
mrs r0, cpsr
bic r0,r0,#0x80
msr cpsr_c,r0 @enabling interrupts in the cpsr
```

Deixando essas instruções após a programação do timer, tudo funciona normalmente.

Se as instruções não forem habilitadas com essas instruções, as interrupções nunca acontecem.

10.6.3 Endereço do vetor de interrupcao

Como o código definiu o vetor de interrupcao? Responda relacionando com o ARM e o ldscript.

O código definiu o endereço do vetor de interrupção a partir do linker script:

```
ENTRY(_Reset)

SECTIONS
{
    . = 0x0;

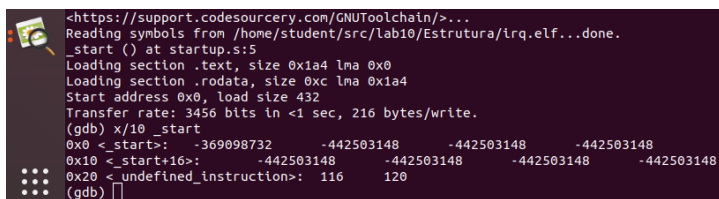
    .text : {
        startup.o (INTERRUPT_VECTOR)
        *(.text)
    }

    .data : { *(.data) }

    .bss : { *(.bss) }

    . = . + 0x1000; /* 4kB of stack memory */

    stack_top = .;
}
```

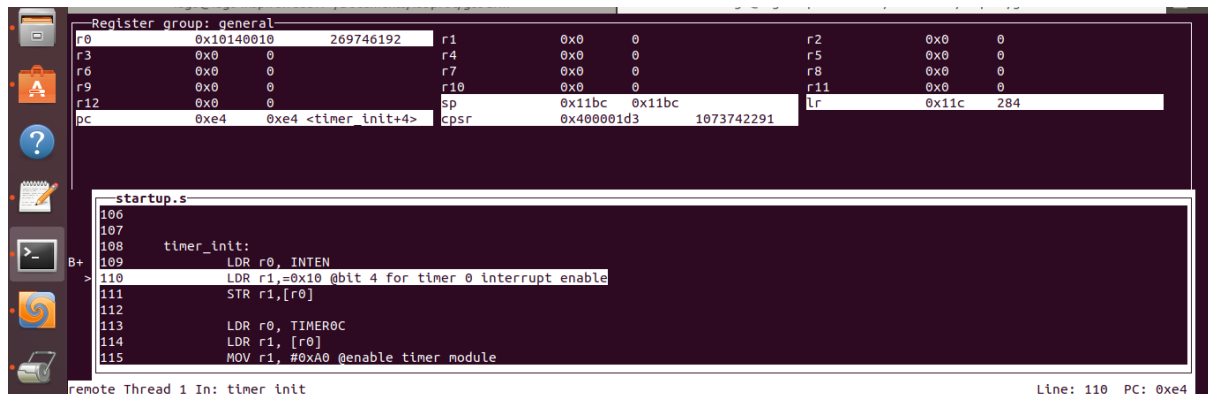


```
<https://support.codesourcery.com/GNUToolchain/>...
Reading symbols from /home/student/src/lab10/Estrutura/irq.elf...done.
_start () at startup.s:5
Loading section .text, size 0x1a4 lma 0x0
Loading section .rodata, size 0xc lma 0x1a4
Start address 0x0, load size 432
Transfer rate: 3456 bits in <1 sec, 216 bytes/write.
(gdb) x/10 _start
0x0 <_start>: -369098732 -442503148 -442503148 -442503148
0x10 <_start+16>: -442503148 -442503148 -442503148 -442503148
0x20 <undefined_instruction>: 116 120
(gdb) □
```

Vemos na imagem que o vetor de interrupções se inicia em 0x0, como descrito no linker script.

10.6.4 Timer, inicialização

Qual é o modo em que o processador roda logo no início enquanto o timer está sendo configurado? Qual registrador deve ser observado? Qual é sp utilizado?



O processador roda em modo Supervisor no início da configuração do timer. Devemos observar o registrador CPSR. Utilizamos o sp do modo Supervisor.

O Mitsuo/2018 pensou no modo e na sequência de programação do timer, de tal forma que a alteração do TIMER 0, altera o intervalo das interrupções. Troque o timer_{init} no seu código pelo código abaixo e faça experiências alterando o valor de TIMER0L.

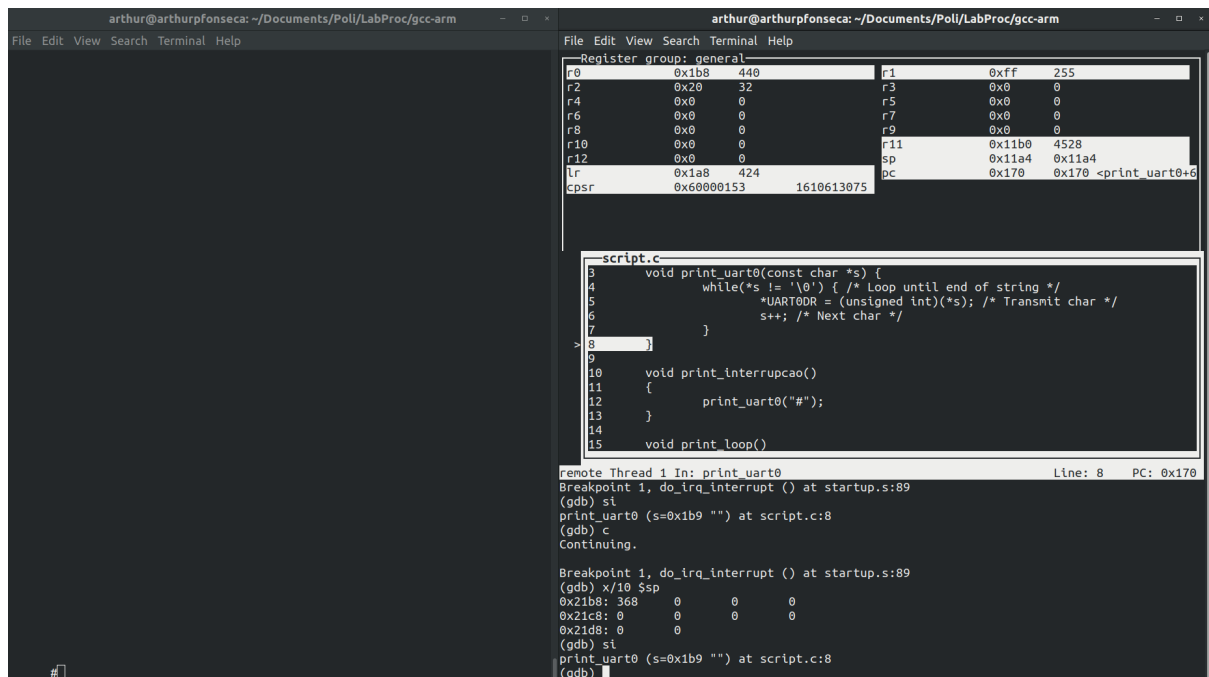
Trocando o código abaixo da direita pelo da esquerda, a interrupção nunca acontece:

<pre>timer_init: LDR r0, INTEN LDR r1,=0x10 @bit 4 for timer 0 interrupt enable STR r1,[r0] LDR r0, TIMER0C LDR r1, [r0] MOV r1, #0xA0 @enable timer module STR r1, [r0] LDR r0, TIMER0V MOV r1, #0xff @setting timer value STR r1,[r0] mrs r0, cpsr bic r0,r0,#0x80 msr cpsr_c,r0 @enabling interrupts in the cpsr</pre>	<pre>timer_init: LDR r0, INTEN LDR r1,=0x10 @bit 4 for timer 0 interrupt enable STR r1,[r0] LDR r0, TIMER0L LDR r1, =0xffffffff @setting timer value STR r1,[r0] LDR r0, TIMER0C MOV r1, #0xE0 @enable timer module STR r1, [r0] mrs r0, cpsr bic r0,r0,#0x80 msr cpsr_c,r0 @enabling interrupts in the cpsr mov pc, lr</pre>
--	---

```
mov pc, lr
```

10.6.5 Fazendo o tratamento da interrupção em C.

Altere o programa principal para que ele fique em um loop continuamente imprimindo o dígito " " (espaço) de tempo em tempo (fique em um loop para gastar tempo). Assim, na tela você deverá ver " "s e "#s" intercalados. **IMPORTANTE:** tire um printscreen dos 's' e '#' sendo impressos para entregar ao professor como atividade.



```
Register group: general
r0 0x1b8 440 r1 0xff 255
r2 0x20 32 r3 0x0 0
r4 0x0 0 r5 0x0 0
r6 0x0 0 r7 0x0 0
r8 0x0 0 r9 0x0 0
r10 0x0 0 r11 0x1b0 4528
r12 0x0 0 sp 0x11a4 0x11a4
lr 0x1a8 424 pc 0x170 0x170 <print_uart0+6>
cpsr 0x60000153 1610613075

script.c
3 void print_uart0(const char *s) {
4     while(*s != '\0') { /* Loop until end of string */
5         *UART0DR = (unsigned int)(*s); /* Transmit char */
6         s++; /* Next char */
7     }
8 }
9
10 void print_interrupcao()
11 {
12     print_uart0("#");
13 }
14
15 void print_loop()

remote Thread 1 In: print_uart0
Breakpoint 1, do_irq_interrupt () at startup.s:89
(gdb) si
print_uart0 (s=0x1b9 "") at script.c:8
(gdb) c
Continuing.

Breakpoint 1, do_irq_interrupt () at startup.s:89
(gdb) x/10 $sp
0x21b8: 0 0 0
0x21c8: 0 0 0
0x21d8: 0 0 0
(gdb) si
print_uart0 (s=0x1b9 "") at script.c:8
(gdb)
```

É possível observar o `#` sendo impresso a cada vez que damos um continue, o programa imprime vários espaços vazios e em seguida um `#` indicando que houve uma interrupção de tempo, como esperado.

O que acontece se não retirarmos o pedido de interrupção na rotina que trata a interrupção? Experimente deixando o programa rodar e explique. A frequência com que os caracteres são impressos varia se não retirarmos o pedido de interrupção? Ou seja, a relação entre " "s e "#s" foi alterada? Explique no relatório. Rode o programa de uma vez sem breakpoints. Em pelo menos uma equipe, aconteceu do resultado ser diferente rodando passo a passo pois nesse caso, o 1 e 2 eram intercalados enquanto que rodando de uma vez o resultado era o esperado (somente a rotina de interrupção imprimia).

Ele sempre imprime "#", pois não sai do modo de irq.

Se não retirar o pedido, a frequência varia significativamente.

Apêndice

10.6.5

handler.c

```
volatile unsigned int * const TIMER0X = (unsigned int *)0x101E200c;
volatile unsigned int * const UART0DR = (unsigned int *)0x101f1000;

void print_uart0(const char *s) {
    while(*s != '\0') { /* Loop until end of string */
        *UART0DR = (unsigned int)(*s); /* Transmit char */
        s++; /* Next char */
    }
}

void hello_world() {
    print_uart0("Hello World! \n");
}

void print_interrupcao() {
    *TIMER0X = 0;
    print_uart0("#");
    return;
}

void space() {
    print_uart0(" ");
    return;
}
```

handler.s

```
.global handler_timer
.text
@TIMER0X: .word 0x101E200c @timer 0 interrupt clear register
handler_timer:
    BL print_interrupcao
    mov pc, lr @retorna
```

irq.s

```
.global _start
.text

_start:
    b _Reset @posição 0x00 - Reset
    ldr pc, _undefined_instruction @posição 0x04 - Instrução não-definida
    ldr pc, _software_interrupt @posição 0x08 - Interrupção de Software
    ldr pc, _prefetch_abort @posição 0x0C - Prefetch Abort
    ldr pc, _data_abort @posição 0x10 - Data Abort
    ldr pc, _not_used @posição 0x14 - Não utilizado
    ldr pc, _irq @posição 0x18 - Interrupção (IRQ)
    ldr pc, _fiq @posição 0x1C - Interrupção(FIQ)

_undefined_instruction: .word undefined_instruction
_software_interrupt: .word software_interrupt
_prefetch_abort: .word prefetch_abort
_data_abort: .word data_abort
_not_used: .word not_used

_irq: .word irq
_fiq: .word fiq

INTPND: .word 0x10140000 @Interrupt status register
INTSEL: .word 0x1014000C @interrupt select register( 0 = irq, 1 = fiq)
INTEN: .word 0x10140010 @interrupt enable register
TIMER0L: .word 0x101E2000 @Timer 0 load register
TIMER0V: .word 0x101E2004 @Timer 0 value registers
TIMER0C: .word 0x101E2008 @timer 0 control register
TIMER0X: .word 0x101E200c @timer 0 interrupt clear register

_Reset:
    MRS r0, cpsr @ salvando o modo corrente em R0
    MSR cpsr_ctl, #0b11010010 @ alterando para modo interrupt
    LDR sp, =timer_stack_top @ a pilha de interrupções de tempo é setada
    MSR cpsr, r0 @ retorna para o modo anterior

    LDR sp, =stack_top

    bl main
    b .

undefined_instruction:
    b .

software_interrupt:
    b do_software_interrupt @vai para o handler de interrupções de software

prefetch_abort:
    b .
```



```

data_abort:
    b .

not_used:
    b .

irq:
    b do_irq_interrupt @vai para o handler de interrupções IRQ

fiq:
    b .

do_software_interrupt: @Rotina de Interrupção de software
    add r1, r2, r3 @r1 = r2 + r3
    mov pc, r14 @volta p/ o endereço armazenado em r14

do_irq_interrupt: @Rotina de interrupções IRQ
    STMFD sp!, {r0 - r3} @Empilha os registradores
    STMFD sp!, {lr}

    LDR r0, INTPND @Carrega o registrador de status de interrupção
    LDR r0, [r0]
    TST r0, #0x0010 @verifica se é uma interrupção de timer

    @MRS r0, cpsr @ salvando o modo corrente em R0
    @MOV r1, sp; @ salva sp antes de alterar

    @MSR cpsr_ctl, #0b11010011 @ alterando para modo 3

    STMFD sp!, {pc} @ salva pc na pilha do modo de interrupções
    @MSR cpsr, r0 @ retorna para o modo anterior

    BLNE handler_timer @vai para o rotina de tratamento da interrupção de timer
    LDMFD sp!, {lr}
    LDMFD sp!, {r0 - r3} @retorna

    sub lr, lr, #4 @ corrigindo o lr
    STMFD sp!, {lr}
    LDMFD sp!, {pc}^
    @mov pc, r14^

timer_init:
    LDR r0, INTEN
    LDR r1, =0x10 @bit 4 for timer 0 interrupt enable
    STR r1, [r0]

    LDR r0, TIMER0C
    LDR r1, [r0]
    MOV r1, #0xA0 @enable timer module
    STR r1, [r0]

```

```
LDR r0, TIMER0V
MOV r1, #0xff @setting timer value
STR r1,[r0]
```

```
mrs r0, cpsr
bic r0,r0,#0x80
msr cpsr_c,r0 @enabling interrupts in the cpsr
```

```
mov pc, lr
```

```
main:
```

```
    bl timer_init @initialize interrupts and timer 0
```

```
stop:
```

```
    BL space
```

```
    b stop
```