FEEC - UNICAMP;
EA871;
EXPERIMENTO DA TAREFA 2;
VINÍCIUS ESPERANÇA MANTOVANI, RA: 247395;

Ítem 1)

a)

Em assembly:

contador++:

- ldr r3, [pc, #52]
- ldrh r3, [r3, #0]
- adds r3, #1
- uxth r2, r3
- ldr r3, [pc, #44]
- strh r2, [r3, #0]

A instrução que carrega o registrador com a variável contador é ldr e, em seguida ldrh para setá-la como half-word no registrador. A instrução adds incrementa o valor de contador. A instrução uxth extende a variável contador de 16 bites para os 32 bits da palavra de memória E, a instrução strh salva o valor incrementado na variável contador.

b)

Partindo da instrução *(uint32_t volatile *) 0x40048038u |= (1<<10); podemos explicar a afirmação pois, entre as instruções em assembly, notamos o uso de endereçamento relativo a PC nas instruções:

- ldr r3, [pc, #104]
- ldr r2, [pc, #100].

Nas quais o conteúdo salvo no registrador parte do endereço de PC mais o valor determinado pelo imediato em seguida.

c)

A linha de comando em C *i = valor * 32; foi traduzida para:

- ldr r3, [r7, #4]
- lsls r2, r3, #5
- ldr r3, [r7, #0]
- str r2, [r3, #0]

A instrução equivalente à multiplicação é lsls r2, r3, #5, na qual é feito um deslocamento dos bits de r3 para a esquerda, o que corresponde a um produto do conteúdo de r3 com 2^5, que é 32. Assim, o valor da multiplicação é salvo em r2. Isso causa um ganho considerável no tempo de execução, uma vez que a operação de multiplicação é custosa em quesito de tempo e processamento, logo, ao evitá-la, substituindo-a por uma operação consideravelmente mais barata de deslocamento de bits, o tempo de execução é decrementado de maneira bastante útil e agradável.

d)

O bloco (1) de instruções é:

- movs r3, #128
- lsls r3, r3, #7
- adds r0, r3, #0

O bloco (2) de instruções é:

- push {r7, lr}
- str r0, [r7, #4]
- strh r2, [r3, #0]
- str r2, [r3, #0]
- str r3, [r7, #12]

O bloco (3) de instruções é:

- mov sp, r7
- add sp, #16
- pop {r7, pc}
- bl 0 <main>

e) Os deslocamentos #104 e #4 são codificados nos últimos 8 bits do código de máquina, sendo necessário adicionar dois 00 à direita desses 8 bits para ler a codificação. Sendo assim, para encontrar o endereço efetivo dos operandos, deve-se adicionar dois zeros à direita dos últimos 8 bits em binário e somar o com conteúdo de PC.

f) Na chamada da função, percebemos que r7 contém o endereço da variável i menos 12 bytes e, que r1 acaba por armazenar o valor de r3 que é, no momento, igual ao valor de r7 + 12, ou seja, é o endereço de i. Além disso, r0 termina por armazenar o valor contido no endereço de valor igual ao r7 mais 4, que contém o valor da variável “valor”. Durante a função, antes da execução das instruções que a compõem, são empilhados os valores de r0 e r1, ou seja, o conteúdo da variável valor e o endereço da variável i. Então, o valor contido em i é definido como produto de “valor” por 32, por meio de um left shift. Para esse processo, são desempilhados o endereço de i e a variável valor. Logo, notamos que o conceito do uso de ponteiros, como passados por parâmetros, se dá de forma que, ao passar o valor da variável em sí, passa-se o valor de seu endereço para o registrador que entra de parâmetro. Enquanto que, durante o uso do *i (valor de i) dentro da função, o que se toma em linguagem de montagem é o conteúdo do que está contido no endereço de i, ou seja, o conteúdo contido no endereço cujo valor está armazenado em um determinado registrador, neste caso r1. Isso permite que as alterações feitas na variável passada por meio de um ponteiro sejam mantidas após o retorno da função.

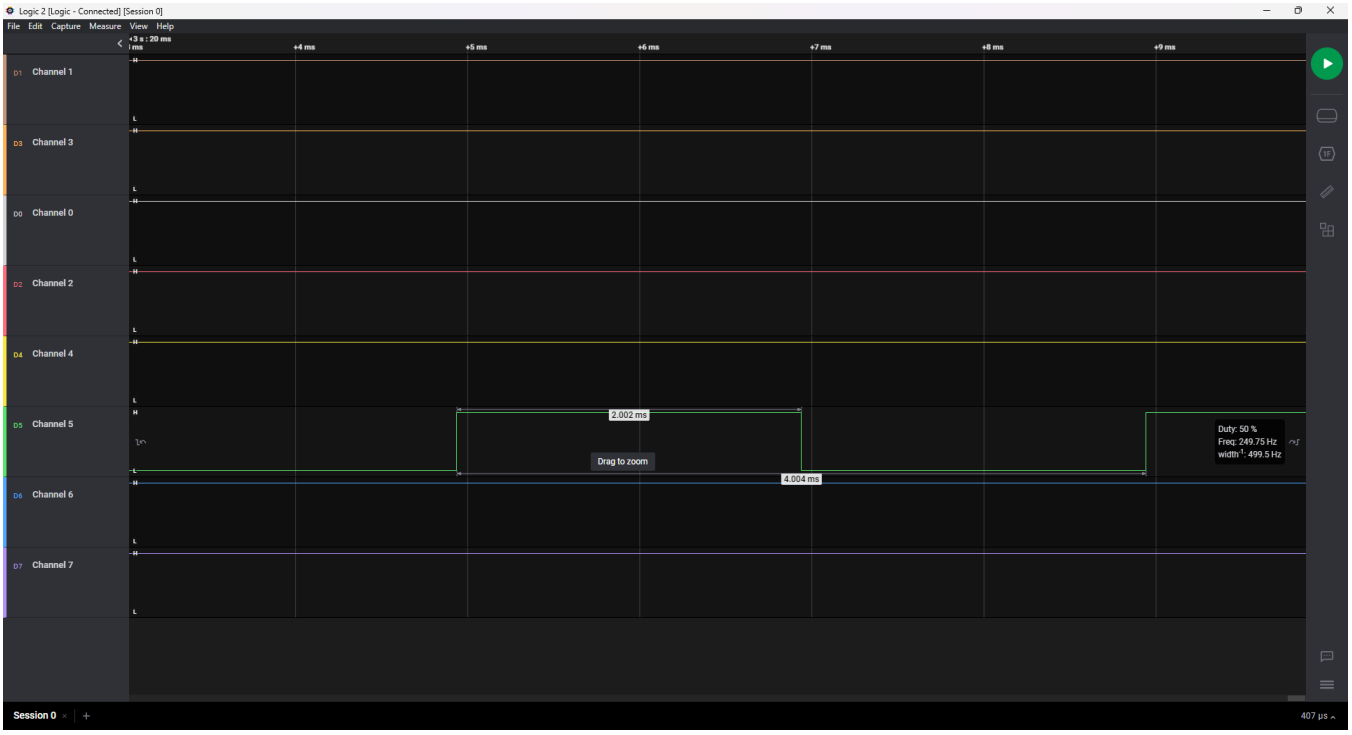
Ítem 2)

a) A segunda solução proposta não foi utilizada, pois, como se explica no manual de referência indicado, a instrução NOP não garante um aumento no tempo de execução, não alterando, por vezes, o tempo ou, até mesmo diminuindo-o. Sendo assim, correr-se-ia o risco de não adicionar a espera desejada na execução.

b)

Instrução	Ciclos de relógio
espera:	10
push {r0,r2,r3,r7,lr}	6
sub sp, sp, #4	1
add r7, sp, #0	1
str r0, [r7,#0]	2
iteracao:	1
mov r2, #NUM_ITERACOES	1
laco:	19 + (bne laco) + (bne interacao)
mov r3, #5	1
orr r3, r0	1
and r3, r0	1
lsr r3, #1	1
asr r3, #1	1
sub r2, #1	1
cmp r2, #0	1
bne laco	1 if not taken, 2 if taken
rev r3, r3	1
lsl r3, #0	1
sub r0, #1	1
cmp r0, #0	1
bne iteracao	1 if not taken, 2 if taken
pop {r0,r2,r3,r7,pc}	8

c) Estimando por meio dos valores anotados na tabela e o número de iterações, chegamos a um valor aproximado de ciclos de 34.011. multiplicando esse número pelo tempo equivalente a um ciclo de relógio, encontramos um valor aproximado de 0,0016214925 s= 1,6214925 ms. Logo, a diferença percentual entre a medição dos sinais no analisador e por meio da estimativa é de aproximadamente 19,0063686%. Uma vez que a medição no analisador é de 2,002 ms.

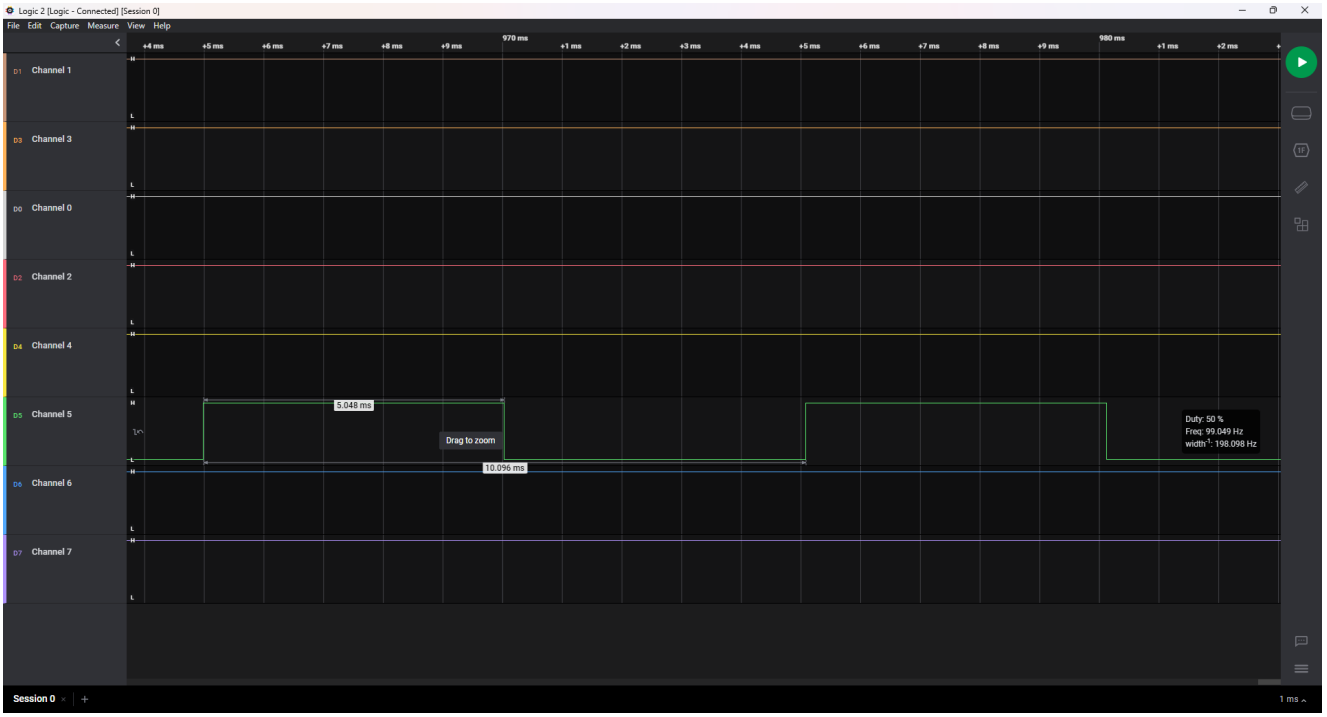


d) usando as instruções em espera conforme as seguintes 5 vezes:

```
mov     r3, #10
orr     r3, r0
and     r3, r0
lsr     r3, #1
asr     r3, #1
```

e a instrução mov r3, #10 uma vez a mais, obtemos um tempo de 5,048 ms no analisador.

Conforme as imagens:




```
espera:                                 @ <delay>
    push {r0,r2,r3,r7,lr}               @ salvar [r0, r2, r3, r7, pc]
    sub sp,sp,#4
    add r7,sp,#0
    str r0,[r7,#0]

iteracao:
    mov r2, #NUM_ITERACOES              @ carregue a qtde de iteracoes
laco:
    mov     r3, #10                         @ operacoes para "gastar" tempo
    orr     r3, r0
    and     r3, r0
    lsr     r3, #1
    asr     r3, #1

    mov     r3, #10                         @ operacoes para "gastar" tempo
    orr     r3, r0
    and     r3, r0
    lsr     r3, #1
    asr     r3, #1

    mov     r3, #10                         @ operacoes para "gastar" tempo
    orr     r3, r0
    and     r3, r0
    lsr     r3, #1
    asr     r3, #1

    mov     r3, #10                         @ operacoes para "gastar" tempo
    orr     r3, r0
    and     r3, r0
    lsr     r3, #1
    asr     r3, #1

    mov     r3, #10                         @ operacoes para "gastar" tempo
```