Planejamento - Experiência 3

Data Processing Operations

Disciplina : PCS3432 - Laboratório de Processadores
Prof: Jorge Kinoshita
Membros:
José Otávio Brochado Colombini - 9795060
Filipe Penna Cerávolo Soares - 10774009

Planejamento **3**

O que há de errado nas seguintes instruções: 3

Sem usar a instrução MUL, de as seguintes instruções para multiplicar o registrador R4 por: 3

Escreva uma rotina que compara 2 valores de 64-bits usando somente 2 instruções. (dica: a segunda instrução é condicionalmente executada, baseada no resultado da primeira comparação). 5

Escreva uma rotina que desloque um valor de 64-bits (armazenado em 2 registradores r0 e r1) de um bit para a direita 5

Idem 4, para a esquerda. 6

Solução de 3.10.7 (individualmente no seu computador) 7

1. Planejamento

a. O que há de errado nas seguintes instruções:

```
ADD r3,r7, #1023
SUB r11, r12, r3, LSL #32
```

O código não pode ser compilado pois as duas constantes são maiores que o permitido, a constante máxima permitida para o ADD devem estar contida em 8 bits ([0, 255]) e a constante para o LSL deve estar contida em 5 bits ([0, 31]) pela arquitetura ARMv7 utilizada na placa do laboratório.

Segue print da resposta do compilador para o comando:

```
student:~/src$ gcc item-3-1-1.s
item-3-1-1.s: Assembler messages:
item-3-1-1.s:0: Warning: end of file not at end of a line; newline inserted
item-3-1-1.s:17: Error: invalid constant -- `add r3,r7,#1023'
item-3-1-1.s:18: Error: invalid immediate shift -- `sub r11,r12,r3,LSL#32'
```

b. Sem usar a instrução MUL, de as seguintes instruções para multiplicar o registrador R4 por:

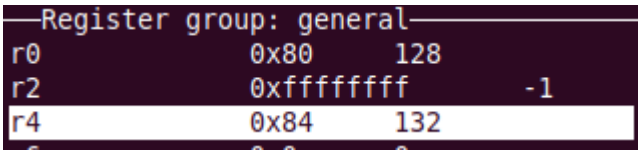
1. 132
2. 255
3. 18
4. 16384

```
main:
    @Multiplicacao sem utilizar o MUL pelos valores indicados nas funcoes
    BL  fun132
    BL  fun255
    BL  fun18
```

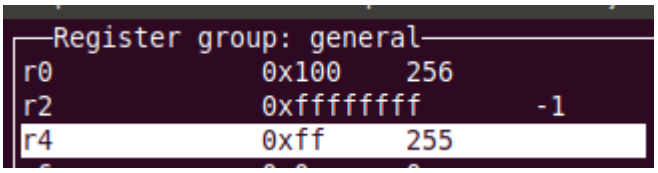
```
    BL  fun16384
    SWI 0x123456

fun132:
    LDR r4, =0x1
    MOV r0, r4, LSL #7
    ADD r4, r0, r4, LSL #2
    MOV pc, lr

fun255:
    LDR r4, =0x1
    MOV r0, r4, LSL #8
    SUB r4, r0, r4
    MOV pc, lr

fun18:
    LDR r4, =0x1
    MOV r0, r4, LSL #4
    ADD r4, r0, r4, LSL #1
    MOV pc, lr

fun16384:
    LDR r4, =0x1
    MOV r4, r4, LSL #14
    MOV pc, lr
```

Segue print dos resultados obtidos:

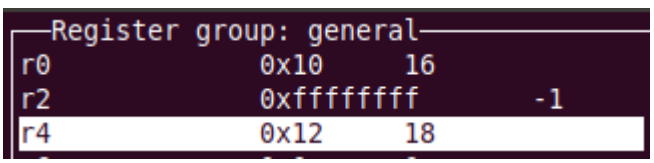
Ao final de fun132:



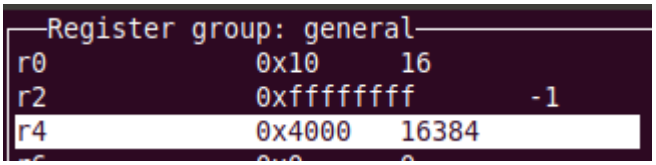
Ao final de fun225:



Ao final de fun18:



Ao final de fun16384:



c. Escreva uma rotina que compara 2 valores de 64-bits usando somente 2 instruções. (dica: a segunda instrução é condicionalmente executada, baseada no resultado da primeira comparação).

```
SUBS r0, r2, r4
SUBEQS r1, r3, r5
```

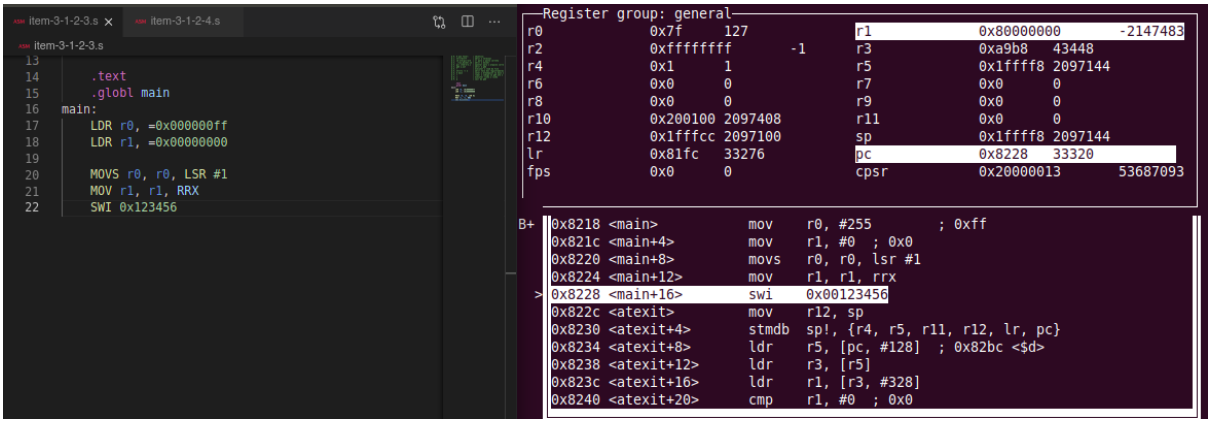
O comando ‘SUBS’ seta a flag ‘Z’, caso r2 = r4. O comando de baixo, por sua vez, é apenas executado caso a flag ‘Z’ esteja setado. Ou seja, apenas de r2 = r4.

d. Escreva uma rotina que desloque um valor de 64-bits (armazenado em 2 registradores r0 e r1) de um bit para a direita

```
LDR r0, =0xFFFFFFFF
LDR r1, =0xFFFFFFFF

MOVS r0, r0, LSR #1
MOV  r1, r1, RRX
SWI  0x123456
```

Considerando que r2 = r0 & r1 que corresponde ao valor de 64 bits, a utilização do RRX permite com que o dado retirado pelo LSR do registrador r0 seja inserido no registrador r1, pois ele fica salvo na flag Carry no CPSR (MOVS é utilizado para atualizar a flag C).

Segue captura de tela representando o valor obtido com essa rotina.

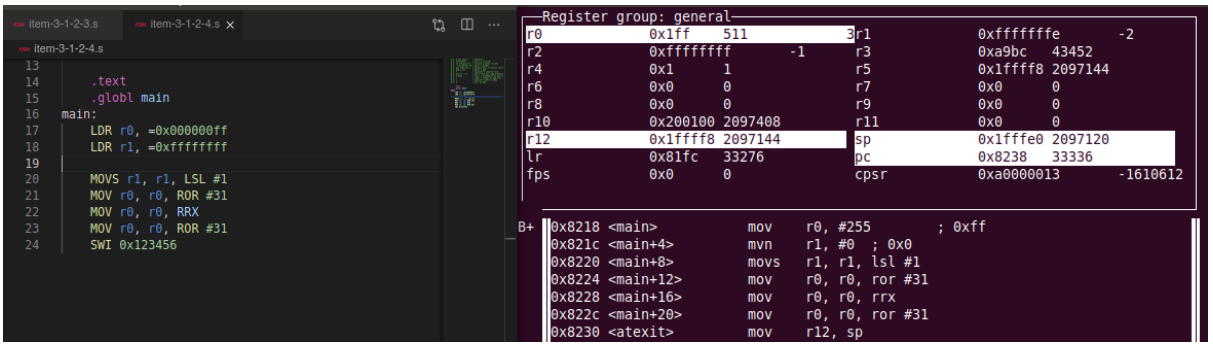


e. Idem 4, para a esquerda.

```
MOVS r1, r1, LSL #1
MOV r0, r0, ROR #31
MOV r0, r0, RRX
MOV r0, r0, ROR #31
SWI 0x123456
```

Novamente, considerando r2 = r0 & r1, utiliza-se o ROR para inverter o registrador e assim inserir o valor do carry, permitido apenas pelo RRX no bit mais significativo. Então para finalizar, o registrador é desinvertido com o ROR novamente.

Segue captura de tela representando o valor obtido com essa rotina.



f. Solução de 3.10.7 (individualmente no seu computador)

```
   LDR r0, =32    @ Dividendo
   LDR r1, =8     @ Divisor
   LDR r2, =0x0     @ Quociente
   LDR r3, =0x0     @ Resto

   @Elementos auxiliares
    LDR r4, =0x80000000 @Identificador de primeiro bit signficativo de
r0 e r1
    LDR r5, =0x80000000 @Identificador de primeiro bit signficativo de
r0 e r1
   LDR r6, =0x1F        @Contador de bit mais significativo de r0
   LDR r7, =0x1F        @Contador de bit mais significativo de r1

   BL  shiftdividend
   BL  opdivision

   MOV r3, r0
   SWI 0x123456

shiftdividend:
@Primeiro verificamos o bit do dividendo
   ANDS     r8, r4, r0
   BNE      shiftdivisor
   MOV      r4, r4,  LSR #1
   SUBS     r6, r6, #1
   B        shiftdividend
shiftdivisor:
@Depois verificamos o do divisor
   ANDS     r8, r5, r1
   MOVNE    pc, lr
   MOV      r5, r5, LSR #1
   SUBS     r7, r7, #1
   B        shiftdivisor

opdivision:
   SUBS r8, r6, r7
   operation:
       MOVPL    r2, r2, LSL #1
       MOVMI    pc, lr
       CMP      r0, r1, LSL r8
       SUBPL    r0, r0, r1, LSL r8
```

```
        ADDPL   r2, r2, #1
        SUBS    r8, r8, #1
        B operation
```

As funções ‘shift dividend’ e ‘shift divisor’ são responsáveis por encontrarem o número de dígitos (em binário) do dividendo e divisor respectivamente.

Com esses valores, ‘opdivision’ executa subtrações com base em shifts para o cálculo do valor em binário para cada uma das casas decimais do quociente.