ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E SISTEMAS DIGITAIS

Relatório 3 de Laboratório de Processador

Turma 02	No. USP
Lucas Haug	10773565
Renzo Armando dos Santos Abensur	10772414

São Paulo

2021

Lucas Haug
Renzo Armando dos Santos Abensur

Relatório 3 de Laboratório de Processador

Relatório apresentado como requisito para avaliação na disciplina PCS3432 - Laboratório de Processadores, no curso de Engenharia Elétrica oferecido pela Escola Politécnica da Universidade de São Paulo.

São Paulo
2021

SUMÁRIO

1. Planejamento

1.1. O que há de errado com as seguintes instruções:

a. ADD r3,r7, #1023

Não foi possível compilar o arquivo com essa soma, pois a instrução ADD só aceita com imediatos de até 500.

```
student:~/src$ gcc item-3.1.2.1.s
item-3.1.2.1.s: Assembler messages:
item-3.1.2.1.s:16: Error: invalid constant -- `add r3,r7,#1023'
```

b. SUB r11, r12, r3, LSL #32

Não foi possível compilar o arquivo com essa subtração, pois o shift imediato na operação de subtração só aceita shifts de no máximo 31 bits e tentou-se utilizar um shift de 32 bits.

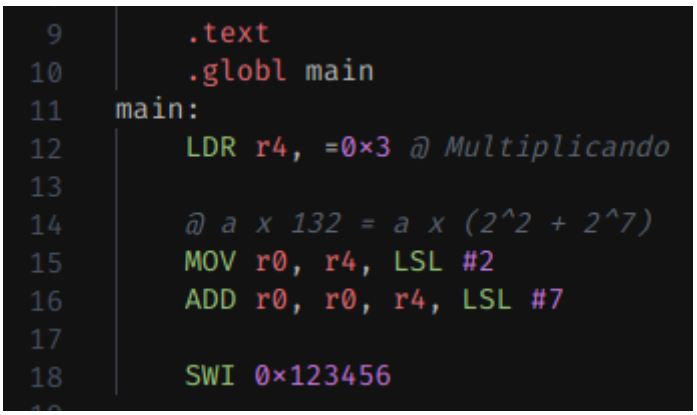
```
student:~/src$ gcc item-3.1.2.1.s
item-3.1.2.1.s: Assembler messages:
item-3.1.2.1.s:16: Error: invalid immediate shift -- `sub r11,r12,r3,LSL#32'
```

2.2. Multiplicação sem usar a instrução MUL

Para o teste das multiplicações desejadas utilizamos o r4 com o valor 0x3.

a.Multiplicação por 132

Para a realização da multiplicação solicitada sem utilizar o MUL primeiramente realizamos MOV r0, r4, LSL #2, ou seja, r0 = r4 * 4 e depois realizamos ADD r0, r0, r4, LSL #7, ou seja, r0 = r0 + r4 * 128. Juntando as duas equações que encontramos temos que r0 = r4 * (4 + 128), portanto fomos capazes de realizar a multiplicação que foi solicitada.

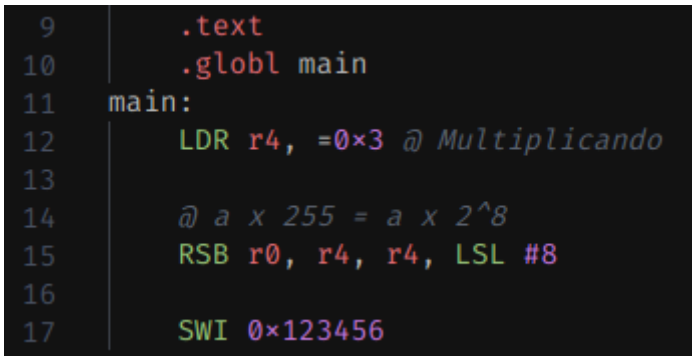


```
9       .text
10      .globl main
11  main:
12      LDR r4, =0x3 @ Multiplicando
13
14      @ a x 132 = a x (2^2 + 2^7)
15      MOV r0, r4, LSL #2
16      ADD r0, r0, r4, LSL #7
17
18      SWI 0x123456
```

Código utilizado

b.Multiplicação por 255

Para a realização da multiplicação solicitada sem utilizar o MUL realizamos RSB r0, r4, r4, LSL #8, ou seja, r0 = r4 * 256 - r4, portanto fomos capazes de realizar a multiplicação r4 * 255.

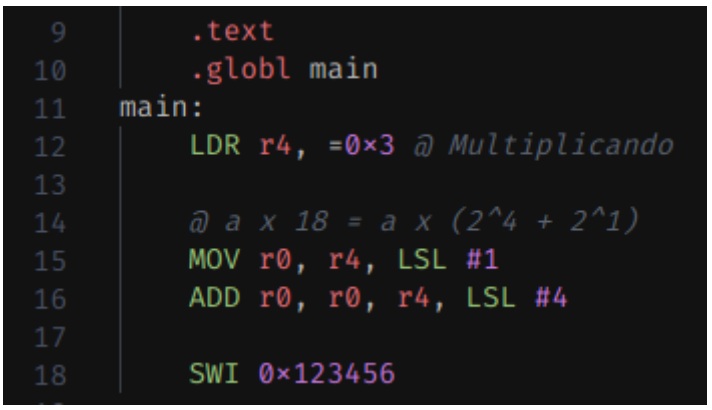


```
9       .text
10      .globl main
11  main:
12      LDR r4, =0x3 @ Multiplicando
13
14      @ a x 255 = a x 2^8
15      RSB r0, r4, r4, LSL #8
16
17      SWI 0x123456
```

Código utilizado

c.Multiplicação por 18

Para a realização da multiplicação solicitada sem utilizar o MUL primeiramente realizamos MOV r0, r4, LSL #1, ou seja, r0 = r4 * 2 e depois realizamos ADD r0, r0, r4, LSL #4, ou seja, r0 = r0 +r4*16. Juntando as duas equações que encontramos temos que r0 = r4 * (2 + 16), portanto fomos capazes de realizar a multiplicação que foi solicitada.

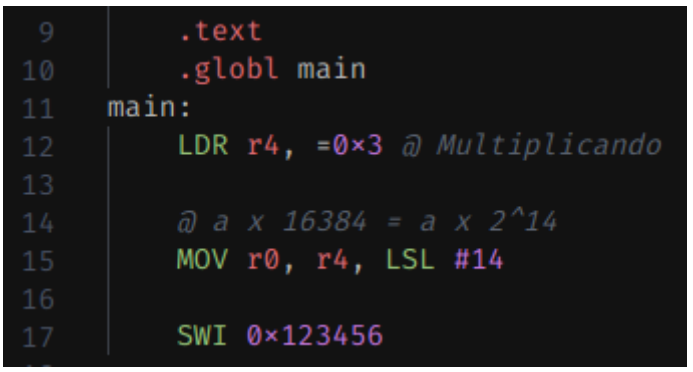


```
9       .text
10      .globl main
11  main:
12      LDR r4, =0x3 @ Multiplicando
13
14      @ a x 18 = a x (2^4 + 2^1)
15      MOV r0, r4, LSL #1
16      ADD r0, r0, r4, LSL #4
17
18      SWI 0x123456
```

Código utilizado

d.Multiplicação por 16384

Para a realização da multiplicação solicitada sem utilizar o MUL bastou fazer um shift de #14.



```
9       .text
10      .globl main
11  main:
12      LDR r4, =0x3 @ Multiplicando
13
14      @ a x 16384 = a x 2^14
15      MOV r0, r4, LSL #14
16
17      SWI 0x123456
```

Código utilizado

3.3. Comparação de dois números de 64 bits

Para comparar 2 valores de 64 bits utilizamos a instrução CMP e a flag EQ. Sendo o primeiro valor a ser comparado colocado nos registradores r0 (parte mais significativa) e r1 (parte menos significativa) e o segundo valor nos registradores r3 (parte mais significativa) e r4 (parte menos significativa). Caso, na primeira comparação, o valor em r1 fosse maior que o valor em r3, então isso indicaria que o valor 1 já é maior que o valor 2 e a segunda comparação nem seria executada. Por outro lado, caso o valor em r1 seja igual ao valor em r3, então a comparação entre o

r2 e r4 seria feita pra determinar qual dos dois números é maior, definindo as flags no CPLR.

```
9       .text
10      .globl main
11  main:
12      MOV r0, #0 @ Registrador auxiliar para ver flags específicas de comparação
13      LDR r1, =0x4 @ Valor 1 [63...32]
14      LDR r2, =0x1 @ Valor 1 [31...0]
15      LDR r3, =0x4 @ Valor 2 [63...32]
16      LDR r4, =0x2 @ Valor 1 [31...0]
17
18      CMP r1, r3 @ Comparção do bloco mais significativo
19      CMPEQ r2, r4 @ Comparção do bloco menos significativo
20
21      MOVLE r0, #1 @ Define o registrador como 1 dependendo da comparação
22
23      SWI 0x123456
```

Código utilizado

4.4. Deslocamento de um valor de 64 bits para direita

Para se fazer o shift para a direita, deve-se fazer primeiramente o shift da parte menos significativa e depois da parte mais significativa, caso haja carry out do shift da parte mais significativa, deve-se adicionar 1 no bit mais significativo da parte menos significativa do número de 64 bits.

```
9       .text
10      .globl main
11  main:
12      LDR r0, =0x11 @ Bits mais significativos do valor para deslocar
13      LDR r1, =0x4 @ Bits menos significativos do valor para deslocar
14
15      MOVS r1, r1, LSR #1
16      MOVS r0, r0, LSR #1
17      ADDCS r1, r1, #0x80000000 @ Soma 1 no bit mais significativo de r1 se houver carry out
18
19      SWI 0x123456
```

Código utilizado

5.5. Deslocamento de um valor de 64-bits para esquerda

Para se fazer o shift para a esquerda, deve-se fazer primeiramente o shift da parte mais significativa e depois da parte menos significativa, caso haja carry out do

shift da parte menos significativa, deve-se adicionar 1 no bit menos significativo da parte mais significativa do número de 64 bits.

```
9       .text
10      .globl main
11  main:
12      LDR r0, =0×11 @ Bits mais significativos do valor para deslocar
13      LDR r1, =0×80000004 @ Bits menos significativos do valor para deslocar
14
15      MOVS r0, r0, LSL #1
16      MOVS r1, r1, LSL #1
17      ADDCS r0, r0, #1 @ Soma 1 no bit menos significativo de r0 se houver carry out
18
19      SWI 0×123456
```

Código utilizado

2. Desenvolvimento

3.3.2 Exercício 3.10.5 - Multiplicação com sinal

Para realizar a multiplicação de 64 bits com sinal utilizando o UMULL utilizamos um registrador r5 para verificar o sinal dos valores a serem multiplicados.

Portanto, inicialmente verificamos se o primeiro número da multiplicação (guardado em r0) é positivo ou negativo com:

CMP r0, #0

Se o número fosse negativo então definiríamos o valor de r5 = 1, caso contrário r5 continua com o valor de 0, além disso se r0 for negativo geramos o valor absoluto dele com:

RSBLT r0, r0, #0

Para o segundo número (guardado em r1), verificamos se ele é negativo com:

CMP r1, #0

Caso for negativo realizamos um XOR de r5 com 1 com o seguinte comando:

EORLT r5, r5, #1

Dessa forma definimos o r5 como 1 caso um dos dois número seja negativo e com 0 caso os dois números sejam positivos ou negativos. Além disso caso o valor no r1 seja negativo geramos o seu valor absoluto com:

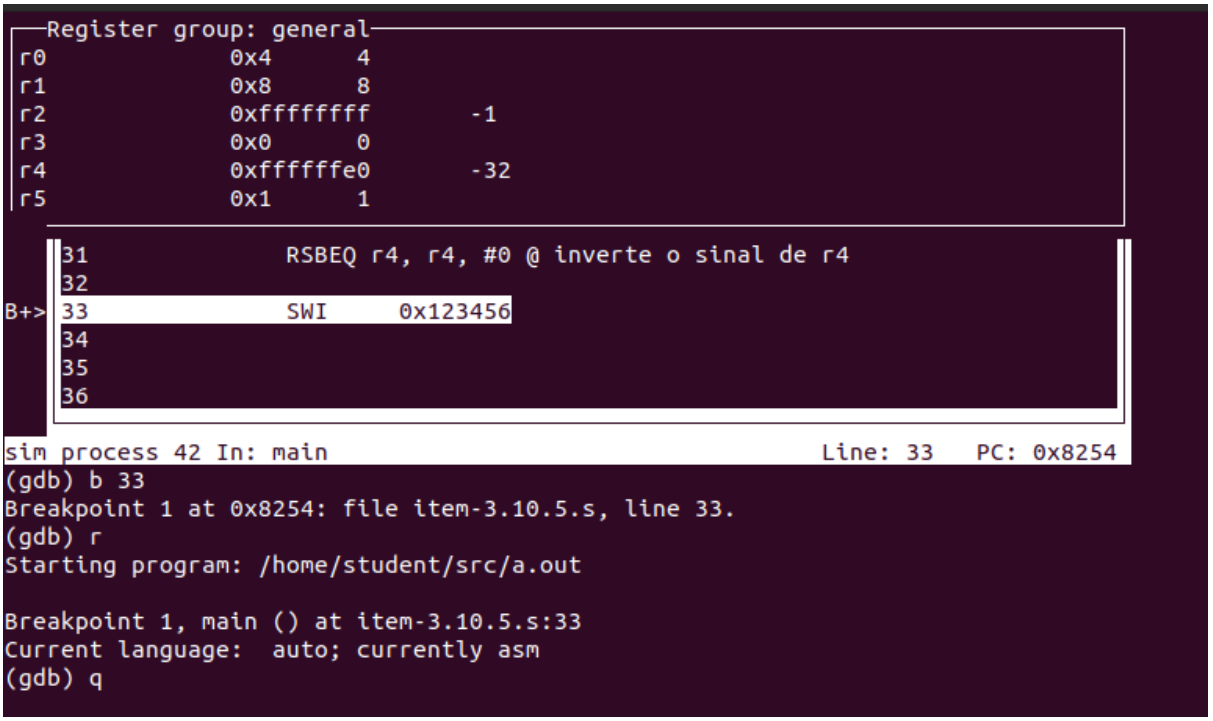
RSBLT r1, r1, #0

Após essas verificações realizamos a multiplicação de r0 e r1 e guardamos o valor em r3 e r4. Por fim, caso o valor de r5 seja 1 isso significa que o valor da multiplicação deve ser negativo e portanto invertemos o sinal final de multiplicação com:

RSBEQ r3, r3
RSBEQ r4, r4.

```
9       .text
10      .globl main
11  main:
12      LDR r0, =4 @ Multiplicando
13      LDR r1, =-8 @ Multiplicador
14      LDR r3, =0x0 @ Resultado [63...32]
15      LDR r4, =0x0 @ Resultado [31...0]
16
17      LDR r5, =0 @ Registrador para verificar se um dos operandos é negativo
18
19      CMP r0, #0
20      LDRLT r5, =1 @ Se r0 < 0, r5 = 1
21      RSBLT r0, r0, #0 @ gera o valor absoluto de r0
22
23      CMP r1, #0
24      EORLT r5, r5, #1 @ Se r1 < 0 r5 = 0 ou r5 = 1 dependendo do valor inicial de r5
25      RSBLT r1, r1, #0 @ gera o valor absoluto de r1
26
27      UMULL r4, r3, r0, r1
28
29      CMP r5, #1
30      RSBEQ r3, r3, #0 @ inverte o sinal de r3
31      RSBEQ r4, r4, #0 @ inverte o sinal de r4
32
33      SWI 0x123456
```

Código utilizado



Resultado

3.3.3 Exercício 3.10.6 - Valor absoluto

Para gerar o valor absoluto de um valor guardado em r0, primeiramente verificamos se o valor é negativo com a comparação de r0 com 0:

CMP r0, #0

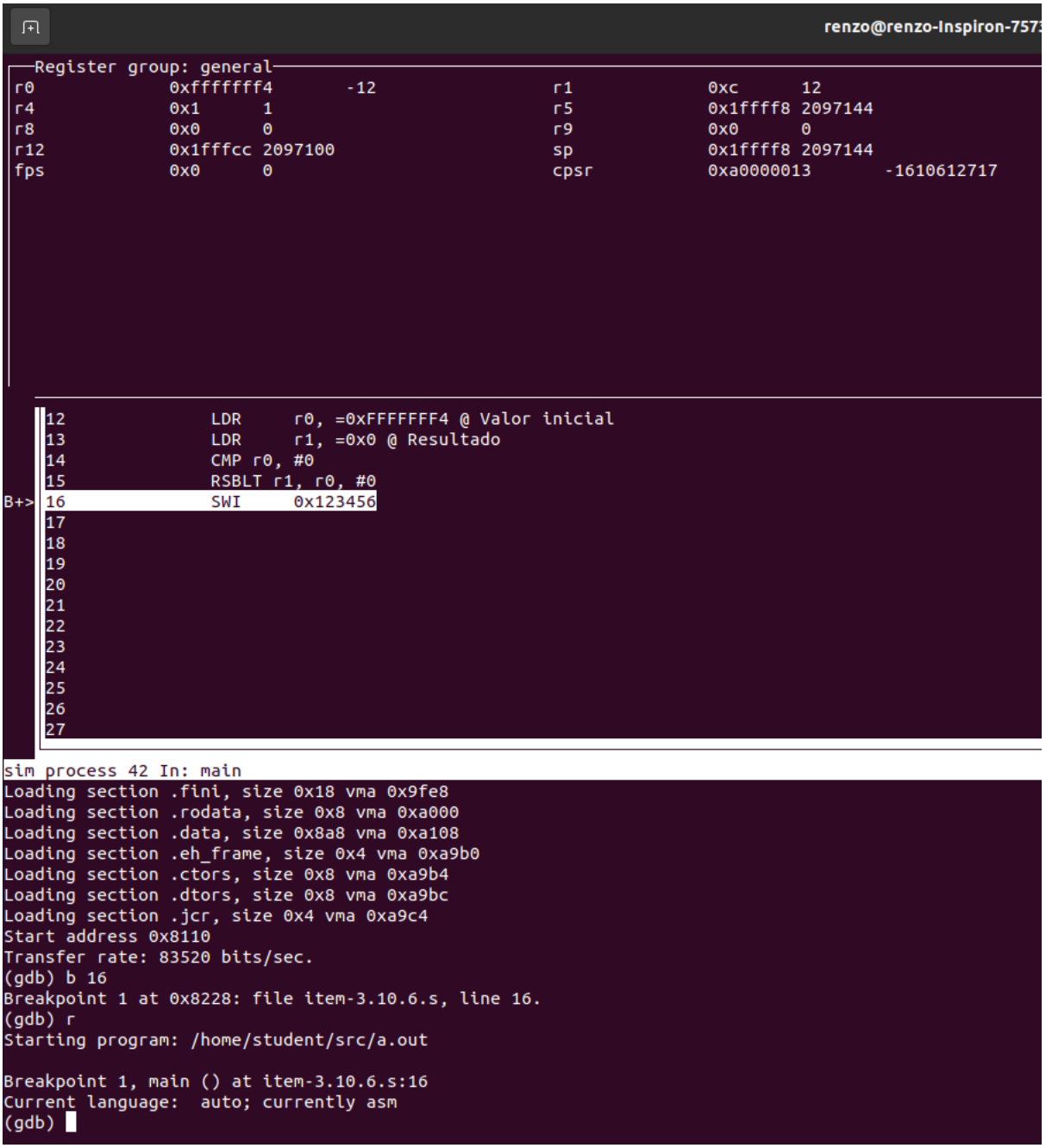
E caso o valor seja menor que 0 armazenamos em r1 o valor positivo de r0 com a operação:

RSBLT r1, r0, #0

A qual representa a subtração r1 = 0 - r1, gerando assim o valor positivo.

```
9       .text
10      .globl main
11  main:
12      LDR r0, =0×FFFFFFF4 @ Valor inicial
13      LDR r1, =0×0 @ Resultado
14      CMP r0, #0
15      RSBLT r1, r0, #0
16      SWI 0×123456
```

Código utilizado



Resultado

3.3.4 Exercício 3.10.7 Divisão

Para se fazer a divisão foi implementado o algoritmo de subtração e deslocamento, para isso primeiramente foi feito o seguinte programa em C para se validar o algoritmo:

```c
#include <stdio.h>
#include <stdint.h>

int main() {
    uint32_t r1 = 0x1F;
    uint32_t r2 = 0x4;
    uint32_t r3 = 0;
    uint32_t r4 = r2;
    uint32_t r5 = 0;

    printf("Dividendo: %d\nDivisor: %d\nQuociente: %d\nResto: %d", r1, r2, r3, r5);

    printf("\n\n");

    while (r1 > r2) {
        r2 = r2 << 1;
    }

    while (r2 >= r4) {
        r3 = r3 << 1;

        if (r1 >= r2) {
            r1 = r1 - r2;
            r3 = r3 + 1;
        }

        r2 = r2 >> 1;
    }

    r5 = r1;

    printf("Dividendo: %d\nDivisor: %d\nQuociente: %d\nResto: %d", r1, r2, r3, r5);

    return 0;
}
```

Algoritmo da divisão em C

Então para se fazer em assembly, inicialmente fazemos uma cópia do valor de r2 (divisor) no registrador r4 com:

MOV r4, r2,

Após essa etapa, criamos um loop para alinhar o divisor com o dividendo com a condição de loop comparando r1 com r2 (CMP r1, r2) e permanecendo no loop enquanto r1 for maior que r2 (BGT allign_loop), dentro do loop é realizado o shit para esquerda de r2.

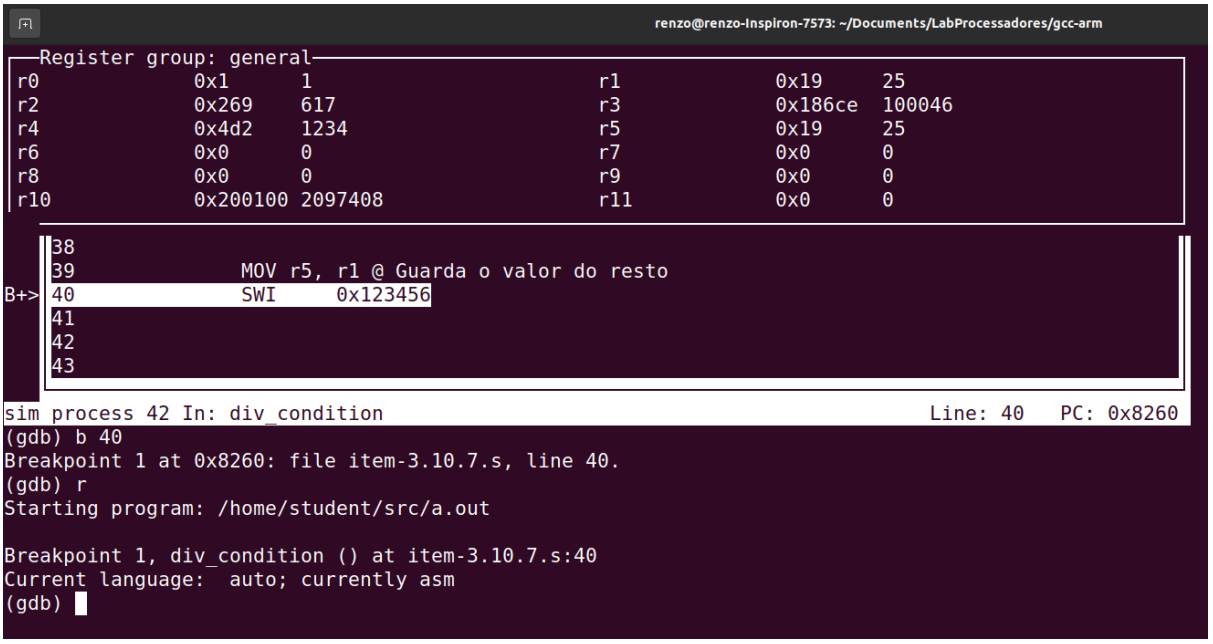
MOV r2, r2, LSL #1.

Na segunda etapa realizamos o loop da divisão que consiste em comparar o valor de r2 e r4, permanecendo no loop enquanto o valor de r2 for maior ou igual ao valor de r4 (BGE div_loop). Dentro do loop primeiramente realizamos o shit para a esquerda do r3 (quociente), depois comparamos r1 com r2 e caso r1 for maior ou

igual a r2 realizamos a subtração do dividendo, r1 = r1 - r2 e a adição do quociente, r3 = r3 + 1, por último realizamos o deslocamento do dividendo para a direita. Após o término do loop da divisão, é armazenado no r5 o valor de r1 que representa o resto da divisão.

```
    .text
    .globl main
main:
    LDR r1, =123456789 @ Dividendo
    LDR r2, =1234 @ Divisor
    LDR r3, =0x0 @ Quociente
    LDR r5, =0x0 @ Resto

    MOV r4, r2

    @ Loop para alinhar o divisor com o dividendo
    B allign_condition
allign_loop:
    MOV r2, r2, LSL #1
allign_condition:
    CMP r1, r2
    BGT allign_loop

    @ Loop que realiza a divisao desejada
    B div_condition
div_loop:
    MOV r3, r3, LSL #1
    CMP r1, r2
    SUBGE r1, r1, r2 @ Subtração do dividendo, r1 = r1 - r2
    ADDGE r3, r3, #1 @ Adição do quociente, r3 = r3 + 1
    MOV r2, r2, LSR #1 @ Deslocamento do dividendo para a direita
div_condition:
    CMP r2, r4
    BGE div_loop

    MOV r5, r1 @ Guarda o valor do resto
    SWI 0x123456
```

Código utilizado


renzo@renzo-Inspiron-7573: ~/Documents/LabProcessadores/gcc-arm
Register group: general
r0 0x1 1 r1 0x19 25
r2 0x269 617 r3 0x186ce 100046
r4 0x4d2 1234 r5 0x19 25
r6 0x0 0 r7 0x0 0
r8 0x0 0 r9 0x0 0
r10 0x200100 2097408 r11 0x0 0
38
39 MOV r5, r1 @ Guarda o valor do resto
B+> 40 SWI 0x123456
41
42
43
sim process 42 In: div_condition Line: 40 PC: 0x8260
(gdb) b 40
Breakpoint 1 at 0x8260: file item-3.10.7.s, line 40.
(gdb) r
Starting program: /home/student/src/a.out
Breakpoint 1, div_condition () at item-3.10.7.s:40
Current language: auto; currently asm
(gdb)

Resultado

Apêndice

EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned
CC or LO	C = 0	Lower, unsigned
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned
LS	C = 0 or Z = 1	Lower or same, unsigned
GE	N = V	Greater than or equal, signed
LT	N != V	Less than, signed
GT	Z = 0 and N = V	Greater than, signed
LE	Z = 1 and N != V	Less than or equal, signed
AL	Can have any value	Always. This is the default when no suffix is specified.