Anaísa Forti da Fonseca, 11811ECP012
Victor Costa Moura Carvalho, 11911ECP003
Lucas Alestério Marques Vieira, 11621ECP016

Projeto Final

Neste projeto final, foi solicitado que implementássemos dois algoritmos: um de multiplicação e outro de divisão. A seguir, mostraremos o fluxo de cada algoritmo desenvolvido, por meio de comentários no código em assembly.

- Multiplicação em ASM

Este algoritmo refere-se ao de multiplicação por adições repetidas. O arquivo asm a seguir foi traduzido para um arquivo hack utilizando os códigos desenvolvidos em aula. Depois, o hack foi executado no CPUEmulator para que fosse possível analisar os resultados. O programa mostrou-se eficiente para 100% dos casos de teste.

Tivemos dificuldade na condição que verificava se B e/ou C eram iguais a 0. De início, tínhamos pensado em fazer uma comparação dos dois juntos por meio de um & (operador lógico “and”). Entretanto, percebemos que wlw só funcionaria perfeitamente com 0s e 1s, e não com todo e qualquer valor que colocássemos ali. Para resolver este problema, separamos a comparação em duas: primeiro víamos se B=0 e depois se C=0. Caso qualquer um deles fosse, era feito um jump para a “ação 1” para seguir corretamente com o programa e definir o produto como 0.

```
(COND1)         // inicia o programa, e verifica primeiro se B é
menor que zero
    @R0
    D=M
    @R4
    M=D         // armazena a variável B original na posição R4
    @R1
    D = M
    @R5
    M=D         // armazena a variável C original na posição R5
    @R0
    D=M
    @R0
    D=M
    @ ACAO2     // se o valor de B (R0) for menor que zero, ele vai
para ação 2.
    D;JLT

(COND2)
    @R1
    D=M
    @ ACAO3     // se o valor de C (R1) for menor que zero, ele vai
```

```
para ação 3.
    D;JLT

(COND3)
    @R0
    D=M         // se B=0, vai para ação 1
    @ ACAO1
    D;JEQ
    @R1
    D=M         // se C=0, vai para a ação 1
    @ ACAO1
    D;JEQ

(MULTIPLIC)     // se ele não caiu em nenhumas das condições
anteriores, realiza a multiplicação normal
    @R0
    D=M
    @ I         // I pega o valor e decrementa
    M=D-1
    @R1
    D=M
    @R2
    D = D+M     // realiza a soma
    M=D         // guarda a soma na posição R2 (produto)
    @ I
    D = M       // pega o valor do I
    @ END1
    D;JEQ       // se o valor for zero dá o JUMP para a função END1
    @R0
    M = D       // atualiza o valor da variável
    @ MULTIPLIC // volta o loop
    0;JMP

(MULTIPLIC2)    // a diferença dessa função para a que se encontra
acima é que nesse caso B ou C é negativo. Logo, ao terminar ele
vai para a função "inverte", para trocar o sinal do produto
    @R0
    D=M
    @ I
    M=D-1
    @R1
    D=M
    @R2
    D = D+M
    M=D
    @ I
    D = M
    @ INVERTE
    D;JEQ
```

```
    @R0
    M = D
    @ MULTIPLIC2
    0;JMP

(ACAO1)         // atribui o valor de A (produto) como 0
    @R2
    M = 0
    @ END
    0; JMP

(ACAO2)         // nessa função, apenas inverte o sinal de B
    @R0
    D = -M
    M = D
    @ MULTIPLIC2// vai para multiplic2, para fazer a inversão do
sinal do produto A depois.
    D;JMP

(ACAO3)         //Nessa função, apenas inverte o sinal de C
    @R1
    D = -M
    M = D
    @ MULTIPLIC2// vai para multiplic2, para inverter o sinal de A
    D;JMP

(INVERTE)       // inverte o sinal de A se B ou C são negativos
    @R2
    M = -M
    @END1
    0;JMP

(END1)          // restaura os valores originais de B e C
    @R4
    D=M
    M=0
    @R0
    M=D
    @R5
    D=M
    M=0
    @R1
    M=D
    @ END
    0; JMP

(END)           // finaliza o programa
    @ END
    0; JMP
```

- Divisão em ASM

Este algoritmo refere-se ao de divisão por subtrações repetidas e sempre produz resto não negativo. O arquivo asm a seguir foi traduzido para um arquivo hack utilizando os códigos desenvolvidos em aula. Despois, o hack foi executado no CPUEmulator para que fosse possível analisar os resultados. O programa mostrou-se eficiente para 100% dos casos de teste.

Tivemos dificuldade em pensar em uma lógica que atendesse à recursividade do programa proposto. No fim, não ouve recursividade, todas as condições e funções foram separadas e desenvolvidas individualmente.

```
    @R0     // armazenando valores originais para recuperar depois
    D=M
    @R5
    M=D
    @R1
    D=M
    @R6
    M=D

    @R1                 // pula para "division by zero" caso o
denominador seja 0
    D=M
    @ DIVISIONBYZERO
    D;JEQ
    @R0                 // pula para "divide" caso o numerador seja
diferente de 0
    D=M
    @ DIVIDE
    D;JNE
    @R2                 // define quociente como 0 caso o numerador
seja 0
    M=0
    @R3                 // define resto como 0 caso o numerador
seja 0
    M=0

(DIVIDE)                // início de "divide"
    @R1
    D=M
    @ COND1
    D;JLT               // pula pra condição 1 caso o denominador
seja menor que 0
    @R0
    D=M
    @ COND2
    D;JLT               // pula pra condição 2 caso o numerador
seja menor que 0
    @ DIVIDEUNSIGNED
```

```
    0;JMP               // pula para "divide unsigned" se
denominador e numerador são maiores que 0

(COND1)                 // início de "condição 1"
    @R1                 // inverte o valor do denominador e volta
para "divide"
    D=M
    D=-D
    M=D
    @ DIVIDE
    0;JMP

(COND2)                 // início de "condição 2"
    @R0                 // inverte o valor do numerador e volta
para "divide"
    D=M
    D=-D
    M=D
    @ DIVIDE
    0;JMP

(DIVIDEUNSIGNED)        // início de "divide unsigned"
    @R2                 // define o quociente como 0
    M=0
    @R0                 // define o resto como o numerador
    D=M
    @R3
    M=D
    (WHILE)             // laço para somar 1 ao quociente enquanto
subtrai o denominador do resto (que agora é o numerador)
        @R3
        D=M
        @R1
        D=D-M
        @ RESTORE
        D;JLT           // caso a condição do laço não seja mais
satisfeita, pula para "restore"
        @R2
        M=M+1
        @R1
        D=M
        @R3
        M=M-D
        @ WHILE
        0;JMP           // volta para o início do laço



(DIVISIONBYZERO)        // início de "division by zero"
```

```
    @R5                 // define o quociente como 0
    M=0
    @R2                 // define o resto como o máximo inteiro
positivo de 16 bits (32767)
    M=0
    @32767
    D=A
    @R3
    M=D
    @ END
    0; JMP              // pula para o fim do programa

(RESTORE)               // início de "restore"
    @R5                 // restaura o valor do numerador para o
original (não invertido)
    D=M
    M=0
    @R0
    M=D
    @ IFELSE
    D;JLT               // pula para "if-
else" caso esse numerador seja menor que 0
    (CONTINUE)
    @R6                 // restaura o valor do denominador para o
original (não invertido)
    D=M
    M=0
    @R1
    M=D
    @ END
    D;JGT               // pula pro fim do programa caso esse
denominador seja maior que 0
    @R2                 // inverte o valor do quociente caso o
denominador seja menor que 0
    M=-M
    @ END
    0;JMP               // pula para o fim do programa

(IFELSE)                // início de "if-else"
    @R3
    D=M
    @ELSE
    D;JNE               // pula para "else" se o resto for
diferente de 0
    @R2                 // inverte o quociente caso o resto seja 0
    M=-M
    @ CONTINUE
    0;JMP               // volta no "restore" para continuar a
restauração dos valores originais
```

```
    (ELSE)              //início de "else"
        @R2             // faz Q = -Q - R
        M=-M
        M=M-1
        @R1             // faz R = D - R
        D=M
        @R3
        M=D-M
        @ CONTINUE
        0;JMP           // volta no "restore" para continuar a
restauração dos valores originais

(END)                   // início de "end"
    @ END
    0;JMP               // loop de fim de programa
```