<1> 10 Points – All of the following ARM assembly language statements contain errors. Errors may include syntax errors, illegal addressing, use of inappropriate registers, invalid constants, etc. Identify in each instruction where the error occurs, and tell me what kind of error is present.

1. MOV R0, Temp
   Use LDR, not MOV (can’t MOV from memory)

2. ADD F1, F0, R2
   Can’t use floating registers with integer ADD

3. ADFS F2, F3, #15
   Floating point constant is out of range (0–5, 10, ½)

4. MOV R4, #513
   Integer constant is out of range (wider than 8 bits)

5. MUL R0, R0, R0
   Destination, first source can’t be same in MUL

6. ADD R0, R0, #1
   Constant is missing # sign (should be #1)

7. LDR LR, R4
   Use MOV, not LDR (can’t LDR from register)

8. SBT R3, R5, R1
   Unknown Op Code

9. ADDGS R0, R0, R6
   Unknown Condition

10. AND R5, R6___
    Missing second operand: AND R5, R6, ___

<2> 10 Points – In each of the following problems you are to multiply the contents of integer register R0 by a constant value, in one instruction, without using any other registers, and without using any explicit multiplication instruction such as MUL, MLA, or UMULL.

1. R0 := R0 × 5
   ADD R0, R0, R0, LSL #2 (R0 + 4×R0)

2. R0 := R0 × 7
   RSB R0, R0, R0, LSL #3 (8×R0 - R0)

3. R0 := R0 × 8
   MOV R0, R0, LSL #3 (8×R0)

4. R0 := R0 × -7
   SUB R0, R0, R0, LSL #3 (R0 - 8×R0)

5. R0 := R0 × -1
   SUB R0, R0, R0, LSL #1 (R0 - 2×R0)
   -or-
   RSB R0, R0, #0 (0 - R0)

<3> 5 Points – Short Essay Answer – You must first assemble and then link your program before loading it into the ARMulator. What is the purpose of the link step?

The link step resolves any symbol addresses left open by the assembler, creating the final runnable binary from one or more assembled blocks. Symbols referenced in one block may be defined in another.
5 Points – Convert the decimal number 7.625 into (a) binary scientific notation (i.e., \( \pm1.xxxx \times 2^Y \)), and (b) the equivalent binary single-precision floating-point representation.

\[ 7.625 = 111.101 = 1.11101 \times 2^2 \] (binary scientific notation)

Biased exponent = 127 + 2 = 129 = 10000001

Mantissa = .11101, and the remainder of the mantissa padded with 0 bits.

Number is positive so sign bit = 0

Final result = 0 10000001 11101000000000000000000

5 Points – Examine the following binary representation of a single-precision floating-point number and show me (a) the equivalent binary scientific notation (i.e., \( \pm1.xxxx \times 2^Y \)) and (b) the final equivalent decimal value.

\[ \text{Sign bit} = 1 \text{ so number is negative.} \]

Biased Exponent = 10000100 = 132, removing bias gives 132 – 127 = 5 true exponent

Mantissa = .1110101, so true fraction = 1.1110101

Binary Scientific Notation = \( -1.1110101 \times 2^5 \)

Binary Fraction = \( -111101.01 \)

Decimal Value = \( -61.25 \)

10 Points – Short Essay Answer – Write a short paragraph comparing the advantages and disadvantages of RISC machines versus CISC machines. Where does each have advantages over the other? Where does each have disadvantages? Give examples where appropriate.

CISC: Lots of functionality in each op code, so a properly designed program will use a few very powerful instructions.

But: The circuitry to implement each instruction is complicated, it may require a lot of “setup” to take advantage of one of these special instructions, and some specialized op codes might never be used. Instructions tend to be variable length.

RISC: Each instruction is very simple and very fast (usually fixed-length, and 1 cycle per instruction), hardware implementation often small and simple, and there are only a few distinct op codes to remember.

But: Doing any significant task may require many more instructions than CISC.
10 Points – In one of our exercises we evaluated the integer polynomial \(2x^2 - 4x + 5\), where the value of \(x\) was in \(R0\) and the result was computed into \(R1\). This time I want you to write a code fragment (not a complete subroutine) to evaluate the same polynomial using \textbf{floating-point numbers}, where the value of \(x\) is in register \(F0\) and the result is to be placed into \(F1\). Do not worry about saving and restoring temporary registers, just compute the result!

\textbf{Solution #1}

\begin{align*}
\text{MUFS F2,F0,F0} & \quad F2=x^2 \\
\text{ADFS F2,F2,F2} & \quad F2=2x^2 \\
\text{MUFS F1,F0,#4.0} & \quad F1=4x \\
\text{RSFS F1,F1,F2} & \quad F1=2x^2-4x \\
\text{ADFS F1,F1,#5.0} & \quad F1=2x^2-4x+5
\end{align*}

\textbf{Solution #2}

\begin{align*}
\text{POWS F1,F0,#2.0} & \quad F1=x^2 \\
\text{MUFS F1,F1,#2.0} & \quad F1=2x^2 \\
\text{MUFS F2,F0,#4.0} & \quad F2=4x \\
\text{SUFS F1,F1,F2} & \quad F1=2x^2-4x \\
\text{ADFS F1,F1,#5.0} & \quad F1=2x^2-4x+5
\end{align*}

15 Points – Trace the following ARM code and show the values of register \(R0\) (in binary) and the flags after each instruction. Write “?” in places where the value is unknowable at the time.

\begin{tabular}{lcccccc}
\textbf{Instructions} & \textbf{N} & \textbf{Z} & \textbf{V} & \textbf{C} & \textbf{R0 (in binary)} \\
\hline
MVNS R0,#0 & 1 & 0 & ? & ? & 11111\ldots 1111 \\
ADDS R0,R0,R0 & 1 & 0 & 0 & 1 & 11111\ldots 1110 \\
ADC R0,R0,#0 & 1 & 0 & 0 & 1 & 11111\ldots 1111 \\
MOV S R0,#3,2 & 1 & 0 & 0 & 1 & 11000\ldots 0000 \\
ADCS R0,R0,R0,ASR #1 & 1 & 0 & 0 & 1 & 10100\ldots 0001 \\
\end{tabular}

Note: the \#3, 2 means “3, right rotate 2 bits”

10 Points – Write the following code fragment in ARM assembly code, using as few instructions as possible.

If \((R0 \text{ is Odd})\) Then \(R1 := R1+R0\)
Else \(R1 := R1-R0\)

For \(R0\) to be odd, its lowest (rightmost) bit must be equal to 1.

\textbf{Solution #1} \hspace{1cm} \textbf{Solution #2}

\begin{tabular}{lcc}
\textbf{No Extra Registers} & \textbf{Uses Extra Register R2} \\
\hline
TST & R0,#1 & ANDS & R2,R0,#1 \\
ADN E & R1,R1,R0 & ADDNE & R1,R1,R0 \\
SUBEQ & R1,R1,R0 & SUBEQ & R1,R1,R0
\end{tabular}
20 Points – Translate the following high-level procedure into a complete, correct, ARM assembly language subroutine. Input parameter \( N \) is to be passed in through the \( R0 \) register. Three ASCII-based ARM subroutines are available, called Print_Blank, Print_Star, and Print_LF (remember that line-feed = ASCII 10), that may be called by your subroutine; all three are completely transparent. The Do-EndDo loop construct shown below runs some fixed number of times without providing an index variable to its loop body; this allows you to write either a count-up loop or a count-down loop depending on which generates the most efficient assembly language. I will be looking for efficiency in your code, so pay particular attention to the overall number of instructions, execution time, register usage, etc. As always, your subroutine must be completely transparent with respect to its register usage, but the only \( \text{LDR/STR} \) instructions you are allowed to use are for saving and restoring registers.

5 Points Extra Credit – What is the shape printed out by this subroutine/procedure?

Procedure Print_Shape(N)
\[
L := 2 \times N - 1 \\
I := 1 \\
\text{While (I <= L) Do} \\
\quad T := \text{Abs}(I - N) \\
\quad \text{Do T Times} \\
\quad \quad \text{Print (" ")} \\
\quad \text{EndDo} \\
\quad \text{Do (L - T) Times} \\
\quad \quad \text{Print ("*")} \\
\quad \quad \text{Print (" ")} \\
\quad \text{EndDo} \\
\quad \text{Print (10)} \\
\quad I := I + 1 \\
\text{EndWhile} \\
\text{EndProcedure}
\]

The printed shape is a hexagon. For example, if \( N = 3 \), then \( L = 5 \) and the shape will be:

\[
\begin{align*}
\text{****} & \quad T=\text{Abs}(1-3) = 2 \text{ blanks, } L-T = 3 \text{ star-blanks} \\
\text{*****} & \quad T=\text{Abs}(2-3) = 1 \text{ blanks, } L-T = 4 \text{ star-blanks} \\
\text{******} & \quad T=\text{Abs}(3-3) = 0 \text{ blanks, } L-T = 5 \text{ star-blanks} \\
\text{*****} & \quad T=\text{Abs}(4-3) = 1 \text{ blanks, } L-T = 4 \text{ star-blanks} \\
\text{****} & \quad T=\text{Abs}(5-3) = 2 \text{ blanks, } L-T = 3 \text{ star-blanks}
\end{align*}
\]
Print_Shape STR LR,SaveLR
STR R1,SaveR1 R1 used as L
STR R2,SaveR2 R2 used as I
STR R3,SaveR3 R3 used as T
STR R4,SaveR4 R4 used as loop ctr

MOV R1,R0,LSL #1
SUB R1,R1,#1 L := 2 * N - 1

MOV R2,#1 I := 1

While1 CMP R2,R1 While (I <= L) Do
BGT EndWhile1

SUBS R3,R2,R0
RSBMI R3,R3,#0 T := Abs(I-N)

MOVS R4,R3 Do T Times
BEQ EndLoop1 (T will be 0)

Loop1 BL Print_Blank Print ("")
SUBS R4,R4,#1
BNE Loop1

EndLoop1 EndDo

SUB R4,R1,R3 Do L - T Times

Loop2 BL Print_Star Print ("*")
BL Print_Blank Print (" ")
SUBS R4,R4,#1
BNE Loop2 EndDo

BL Print_LF Print (10)

ADD R2,R2,#1 I := I + 1
B While1

EndWhile1 EndWhile

LDR R4,SaveR4
LDR R3,SaveR3
LDR R2,SaveR2
LDR R1,SaveR1
LDR PC,SaveLR Return

SaveLR DCD 0
SaveR1 DCD 0
SaveR2 DCD 0
SaveR3 DCD 0
SaveR4 DCD 0

No need to save and restore R0 since it never changes (even though the Print_Star and other routines may use R0 internally, they are known to be completely transparent)