EC-206
Microprocessors
Assignment-2

Utkarsh Mahajan

201EC164

1. Explain ARM Application Procedure Call Standard (AAPCS) for the ARM architecture.

The **AAPCS** (ARM Application Procedure Call Standard) defines how subroutines can be separately written, separately compiled, and separately assembled to work together. It describes a contract between a calling routine and a called routine that defines:

- Obligations on the caller to create a program state in which the called routine may start to execute.
- Obligations on the called routine to preserve the program state of the caller across the call.
- The rights of the called routine to alter the program state of its caller.

This standard specifies the base for a family of Procedure Call Standard (PCS) variants generated by choices that reflect alternative priorities among:

- Code size.
- Performance.
- Functionality (for example, ease of debugging, run-time checking, support for shared libraries).

Some aspects of each variant – for example the allowable use of R9 – are determined by the execution environment. Thus:

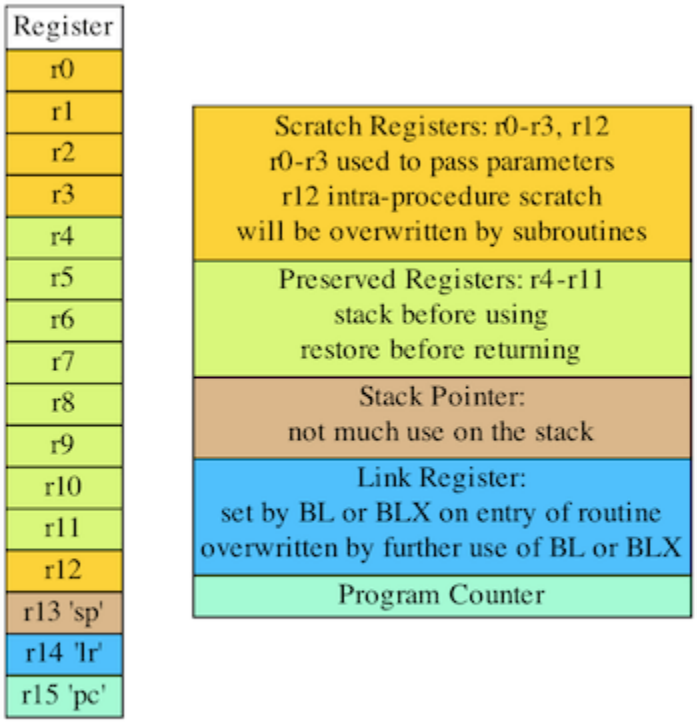
- It is possible for code complying strictly with the base standard to be PCS compatible with each of the variants.
- It is unusual for code complying with a variant to be compatible with code complying with any other variant.
- Code complying with a variant, or with the base standard, is not guaranteed to be compatible with an execution environment that requires those standards. An execution environment may make further demands beyond the scope of the procedure call standard.

This standard is presented in four sections that, after an introduction, specify:

- The layout of data.
- Layout of the stack and calling between functions with public interfaces.
- Variations available for processor extensions, or when the execution environment restricts the addressing model.
- The C and C++ language bindings for plain data types.

This specification does not standardize the representation of publicly visible C++-language entities that are not also C language entities and it places no requirements

on the representation of language entities that are not visible across public interfaces.



Register Use in the ARM Procedure Call Standard

The AAPCS embodies the fifth major revision of the APCS (ARM Procedure Call Standard) and third major revision of the TPCS. It forms part of the complete ABI (Application Binary Interface) specification for the ARM Architecture.

Design Goals:

The goals of the AAPCS are to:

- Support Thumb-state and ARM-state equally.
- Support inter-working between Thumb-state and ARM-state.
- Support efficient execution on high-performance implementations of the ARM Architecture.
- Clearly distinguish between mandatory requirements and implementation discretion.
- Minimize the binary incompatibility with the ATPCS.

Conformance:

The AAPCS defines how separately compiled and separately assembled routines can work together. There is an externally visible interface between such routines. It is

common that not all the externally visible interfaces to software are intended to be publicly visible or open to arbitrary use. In effect, there is a mismatch between the machine-level concept of external visibility—defined rigorously by an object code format—and a higher level, application-oriented concept of external visibility—which is system-specific or application-specific.

Conformance to the AAPCS requires that:

- At all times, stack limits and basic stack alignment are observed.
- At each call where the control transfer instruction is subject to a BL-type relocation at static link time, rules on the use of IP are observed.
- The routines of each publicly visible interface conform to the relevant procedure call standard variant.
- The data elements2 of each publicly visible interface conform to the data layout rules.

THE BASE PROCEDURE CALL STANDARD

The base standard defines a machine-level, core-registers-only calling standard common to the ARM and Thumb instruction sets. It should be used for systems where there is no floating-point hardware, or where a high degree of inter-working with Thumb code is required.

Machine Registers

The ARM architecture defines a core instruction set plus a number of additional instructions implemented by coprocessors. The core instruction set can access the core registers and co-processors can provide additional registers which are available for specific operations.

Core registers

There are 16, 32-bit core (integer) registers visible to the ARM and Thumb instruction sets. These are labeled r0- r15 or R0-R15. Register names may appear in assembly language in either upper case or lower case. In this specification upper case is used when the register has a fixed role in the procedure call standard. In addition to the core registers there is one status register (CPSR) that is available for use in conforming code.

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Table 2, Core registers and AAPCS usage

The first four registers r0-r3 (a1-a4) are used to pass argument values into a subroutine and to return a result value from a function. They may also be used to hold intermediate values within a routine (but, in general, only between subroutine calls).

Register r12 (IP) may be used by a linker as a scratch register between a routine and any subroutine it calls. It can also be used within a routine to hold intermediate values between subroutine calls.

The role of register r9 is platform specific. A virtual platform may assign any role to this register and must document this usage. For example, it may designate it as the static base (SB) in a position-independent data model, or it may designate it as the thread register (TR) in an environment with thread-local storage. The usage of this register may require that the value held is persistent across all calls. A virtual platform that has no need for such a special register may designate r9 as an additional callee-saved variable register, v6.

Typically, the registers r4-r8, r10 and r11 (v1-v5, v7 and v8) are used to hold the values of a routine's local variables. Of these, only v1-v4 can be used uniformly by the whole Thumb instruction set, but the AAPCS does not require that Thumb code only use those registers.

A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP (and r9 in PCS variants that designate r9 as v6).

In all variants of the procedure call standard, registers r12-r15 have special roles. In these roles they are labeled IP, SP, LR and PC.

Processes, Memory and the Stack

The AAPCS applies to a single thread of execution or process (hereafter referred to as a process). A process has a program state defined by the underlying machine registers and the contents of the memory it can access. The memory a process can access, without causing a run-time fault, may vary during the execution of the process.

The memory of a process can normally be classified into five categories:

- code (the program being executed), which must be readable, but need not be writable, by the process.
- read-only static data.
- writable static data.
- the heap.
- the stack.

Writable static data may be further sub-divided into initialized, zero-initialized and uninitialized data. Except for the stack there is no requirement for each class of memory to occupy a single contiguous region of memory. A process must always have some code and a stack, but need not have any of the other categories of memory.

The heap is an area (or areas) of memory that are managed by the process itself (for example, with the C malloc function). It is typically used for the creation of dynamic data objects.

A conforming program must only execute instructions that are in areas of memory designated to contain code.

Subroutine Calls

Both the ARM and Thumb instruction sets contain a primitive subroutine call instruction, BL, which performs a branch-with-link operation. The effect of executing BL is to transfer the sequentially next value of the program counter—the return address—into the link register (LR) and the destination address into the program

counter (PC). Bit 0 of the link register will be set to 1 if the BL instruction was executed from Thumb state, and to 0 if executed from ARM state. The result is to transfer control to the destination address, passing the return address in LR as an additional parameter to the called subroutine.

Control is returned to the instruction following the BL when the return address is loaded back into the PC.

Result Return

The manner in which a result is returned from a function is determined by the type of that result. For the base standard:

A Half-precision Floating Point Type is converted to single precision and returned in r0.

A Fundamental Data Type that is smaller than 4 bytes is zero- or sign-extended to a word and returned in r0.

A word-sized Fundamental Data Type (e.g., int, float) is returned in r0.

A double-word sized Fundamental Data Type (e.g., long long, double and 64-bit containerized vectors) is returned in r0 and r1.

A 128-bit containerized vector is returned in r0-r3.

A Composite Type not larger than 4 bytes is returned in r0. The format is as if the result had been stored in memory at a word-aligned address and then loaded into r0 with an LDR instruction. Any bits in r0 that lie outside the bounds of the result have unspecified values.

A Composite Type larger than 4 bytes, or whose size cannot be determined statically by both caller and callee, is stored in memory at an address passed as an extra argument when the function was called .

Parameter Passing

The base standard provides for passing arguments in core registers (r0-r3) and on the stack. For subroutines that take a small number of parameters, only registers are used, greatly reducing the overhead of a call. Parameter passing is defined as a two-level conceptual model

A mapping from a source language argument onto a machine type

The marshalling of machine types to produce the final parameter list

The mapping from the source language onto the machine type is specific for each language. The result is an ordered list of arguments that are to be passed to the subroutine.

In the following description there are assumed to be a number of co-processors available for passing and receiving arguments. The co-processor registers are divided into different classes. An argument may be a candidate for at most one co-processor register class. An argument that is suitable for allocation to a coprocessor register is known as a Co-processor Register Candidate (CPRC).

In the base standard there are no arguments that are candidates for a co-processor register class.

A variadic function is always marshaled as for the base standard.

For a caller, sufficient stack space to hold stacked arguments is assumed to have been allocated prior to marshaling: in practice the amount of stack space required cannot be known until after the argument marshalling has been completed. A callee can modify any stack space used for receiving parameter values from the caller.

When a Composite Type argument is assigned to core registers (either fully or partially), the behavior is as if the argument had been stored to memory at a word-aligned (4-byte) address and then loaded into consecutive registers using a suitable load-multiple instruction.

Interworking

The AAPCS requires that all sub-routine call and return sequences support inter-working between ARM and Thumb states.

2. ARM Application Procedure Call Create a table in memory having entries for 20 students. Each entry consists of Roll Number (1byte) and Marks (1byte). The maximum mark is decimal #100. Write an assembly program to

(I) Find the maximum and minimum marks and the corresponding roll number (save both in some memory location)

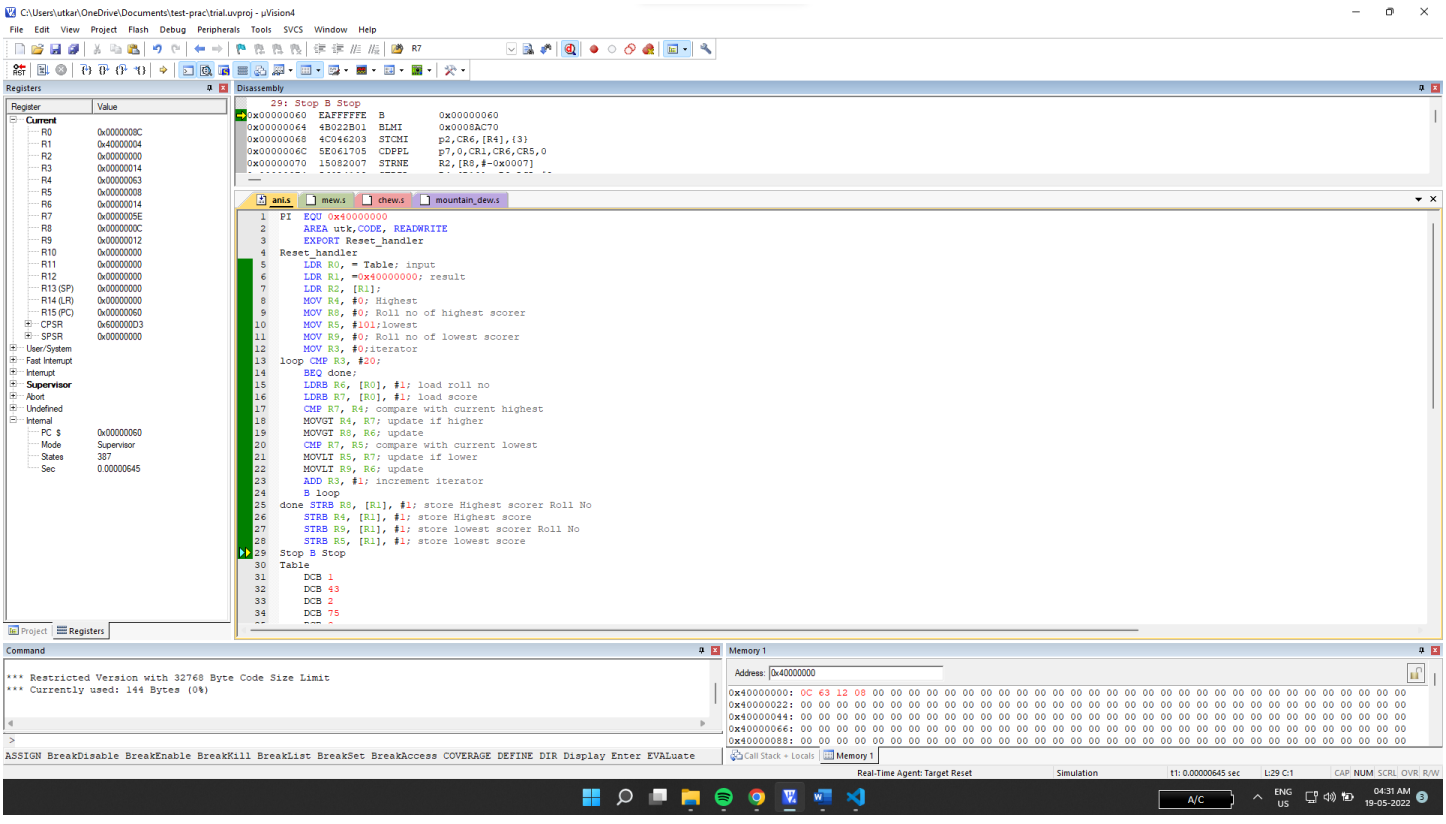
Code:

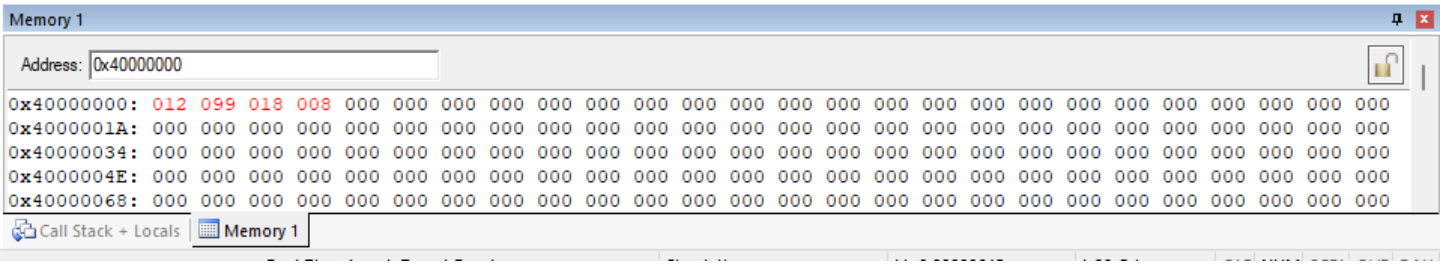
```
    AREA utk,CODE, READWRITE
    EXPORT Reset_handler
Reset_handler
    LDR R0, = Table; input
    LDR R1, =0x40000000; result
    LDR R2, [R1];
    MOV R4, #0; Highest
    MOV R8, #0; Roll no of highest scorer
```

```
    MOV R5, #101;lowest
    MOV R9, #0; Roll no of lowest scorer
    MOV R3, #0;iterator
loop CMP R3, #20;
    BEQ done;
    LDRB R6, [R0], #1; load roll no
    LDRB R7, [R0], #1; load score
    CMP R7, R4; compare with current highest
    MOVGT R4, R7; update if higher
    MOVGT R8, R6; update
    CMP R7, R5; compare with current lowest
    MOVLT R5, R7; update if lower
    MOVLT R9, R6; update
    ADD R3, #1; increment iterator
    B loop
done STRB R8, [R1], #1; store Highest scorer Roll No
    STRB R4, [R1], #1; store Highest score
    STRB R9, [R1], #1; store lowest scorer Roll No
    STRB R5, [R1], #1; store lowest score
Stop B Stop
Table
    DCB 1
    DCB 43
    DCB 2
    DCB 75
    DCB 3
    DCB 98
    DCB 4
    DCB 76
    DCB 5
    DCB 23
    DCB 6
    DCB 94
    DCB 7
    DCB 32
    DCB 8
    DCB 21
    DCB 9
    DCB 65
    DCB 10
    DCB 86
    DCB 11
    DCB 73
```

```
    DCB 12
    DCB 99
    DCB 13
    DCB 94
    DCB 14
    DCB 74
    DCB 15
    DCB 38
    DCB 16
    DCB 84
    DCB 17
    DCB 30
    DCB 18
    DCB 08
    DCB 19
    DCB 65
    DCB 20
    DCB 94
    END
```

Output:





Looking at the memory output window in decimal mode. We can see that the score 99 of roll number 12 is highest and the score 8 of roll no 18 is lowest which can be confirmed from the code above.

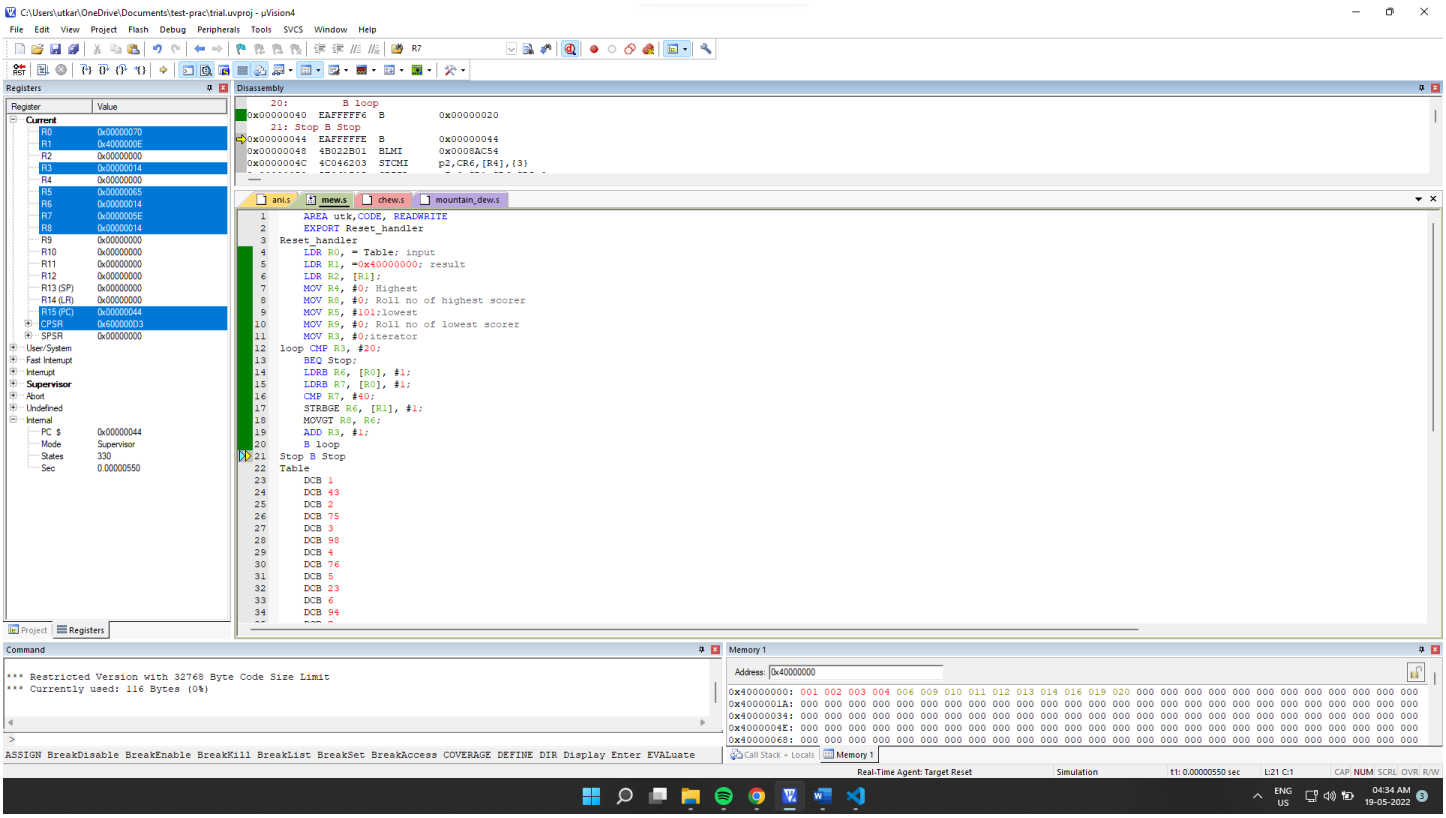
(ii) If passing mark is decimal #40. Find the roll numbers who are passed and save those roll numbers in some memory location.

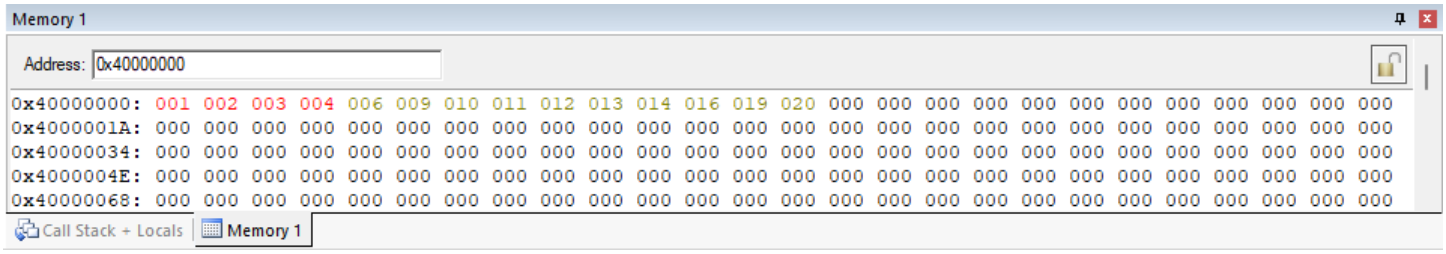
Code:

```
    AREA utk,CODE, READWRITE
    EXPORT Reset_handler
Reset_handler
    LDR R0, = Table; input
    LDR R1, =0x40000000; result
    LDR R2, [R1];
    MOV R4, #0; Highest
    MOV R8, #0; Roll no of highest scorer
    MOV R5, #101;lowest
    MOV R9, #0; Roll no of lowest scorer
    MOV R3, #0;iterator
loop CMP R3, #20;
    BEQ Stop;
    LDRB R6, [R0], #1;
    LDRB R7, [R0], #1;
    CMP R7, #40; check if passed
    STRBGE R6, [R1], #1; store roll no if passed
    ADD R3, #1; increment iterator
    B loop
Stop B Stop
Table
    DCB 1
    DCB 43
    DCB 2
    DCB 75
    DCB 3
    DCB 98
    DCB 4
```

```
    DCB 76
    DCB 5
    DCB 23
    DCB 6
    DCB 94
    DCB 7
    DCB 32
    DCB 8
    DCB 21
    DCB 9
    DCB 65
    DCB 10
    DCB 86
    DCB 11
    DCB 73
    DCB 12
    DCB 99
    DCB 13
    DCB 94
    DCB 14
    DCB 74
    DCB 15
    DCB 38
    DCB 16
    DCB 84
    DCB 17
    DCB 30
    DCB 18
    DCB 08
    DCB 19
    DCB 65
    DCB 20
    DCB 94
    END
```

Output:





Output looking at the output memory window we can verify that the roll numbers saved in the memory are the ones with passing marks.

(iii)Find the average of the class.

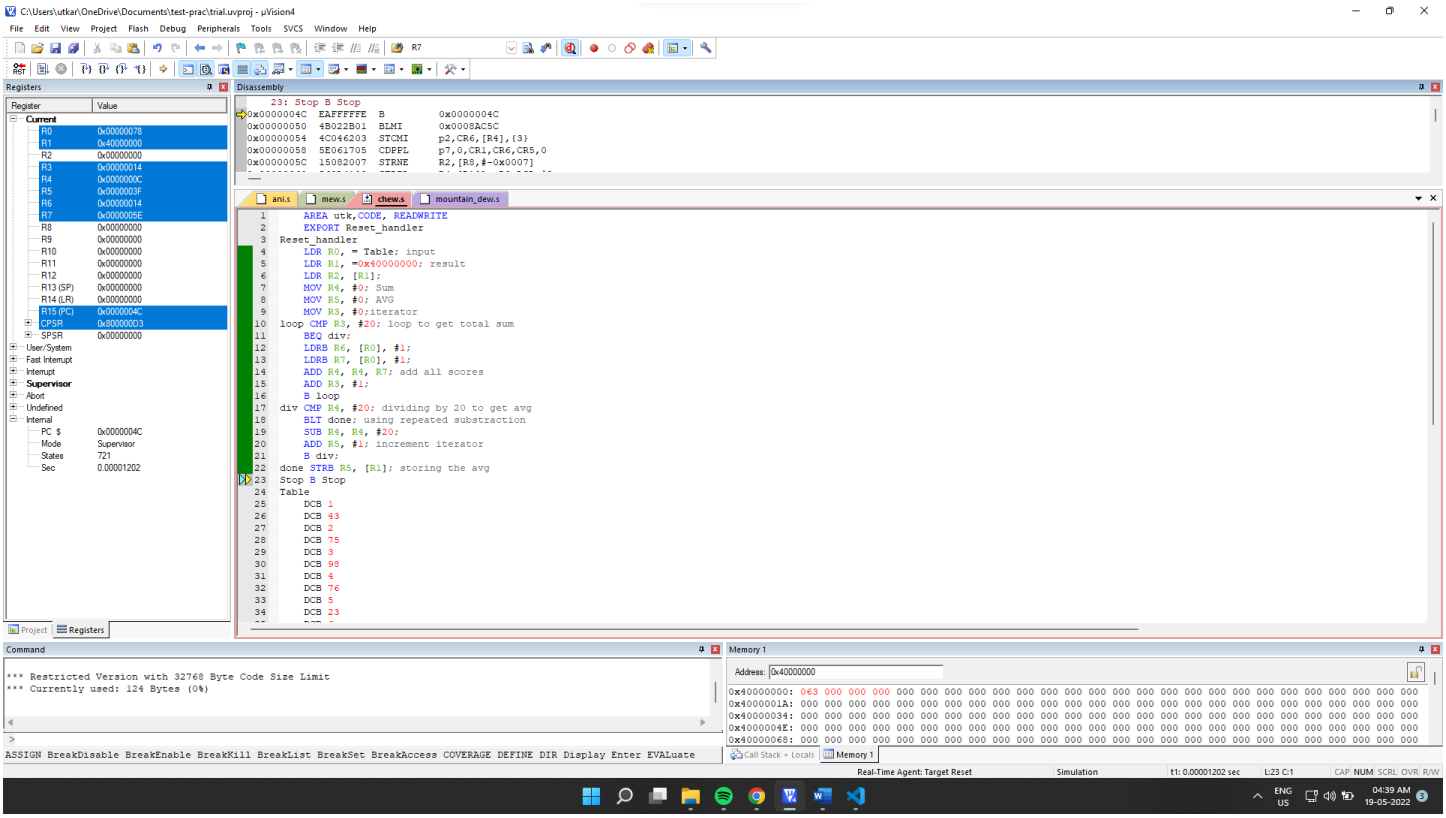
Code:

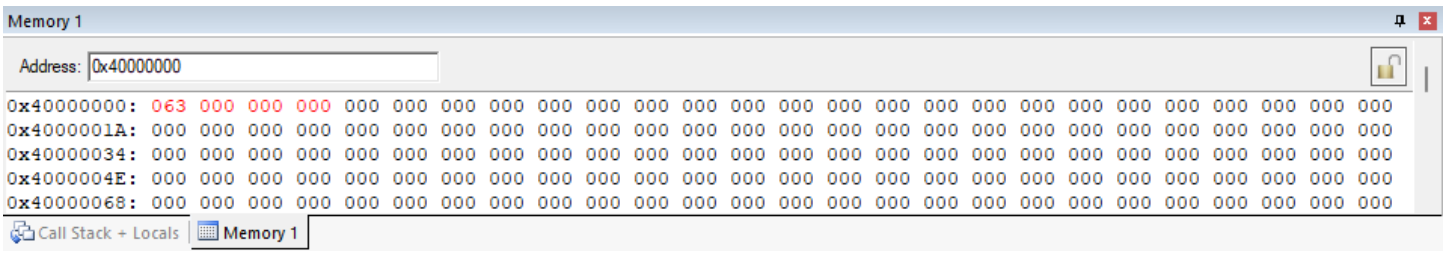
```
    AREA utk,CODE, READWRITE
    EXPORT Reset_handler
Reset_handler
    LDR R0, = Table; input
    LDR R1, =0x40000000; result
    LDR R2, [R1];
    MOV R4, #0; Sum
    MOV R5, #0; AVG
    MOV R3, #0;iterator
loop CMP R3, #20; loop to get total sum
    BEQ div;
```

```
    LDRB R6, [R0], #1;
    LDRB R7, [R0], #1;
    ADD R4, R4, R7; add all scores
    ADD R3, #1;
    B loop
div CMP R4, #20; dividing by 20 to get avg
    BLT done; using repeated substraction
    SUB R4, R4, #20;
    ADD R5, #1; increment iterator
    B div;
done STRB R5, [R1]; storing the avg
Stop B Stop
Table
    DCB 1
    DCB 43
    DCB 2
    DCB 75
    DCB 3
    DCB 98
    DCB 4
    DCB 76
    DCB 5
    DCB 23
    DCB 6
    DCB 94
    DCB 7
    DCB 32
    DCB 8
    DCB 21
    DCB 9
    DCB 65
    DCB 10
    DCB 86
    DCB 11
    DCB 73
    DCB 12
    DCB 99
    DCB 13
    DCB 94
    DCB 14
    DCB 74
    DCB 15
    DCB 38
```

```
    DCB 16
    DCB 84
    DCB 17
    DCB 30
    DCB 18
    DCB 08
    DCB 19
    DCB 65
    DCB 20
    DCB 94
    END
```

Output:





Looking at the memory window In decimal mode, we can see that the result, 63 is the average of the scores of the 20 roll numbers.

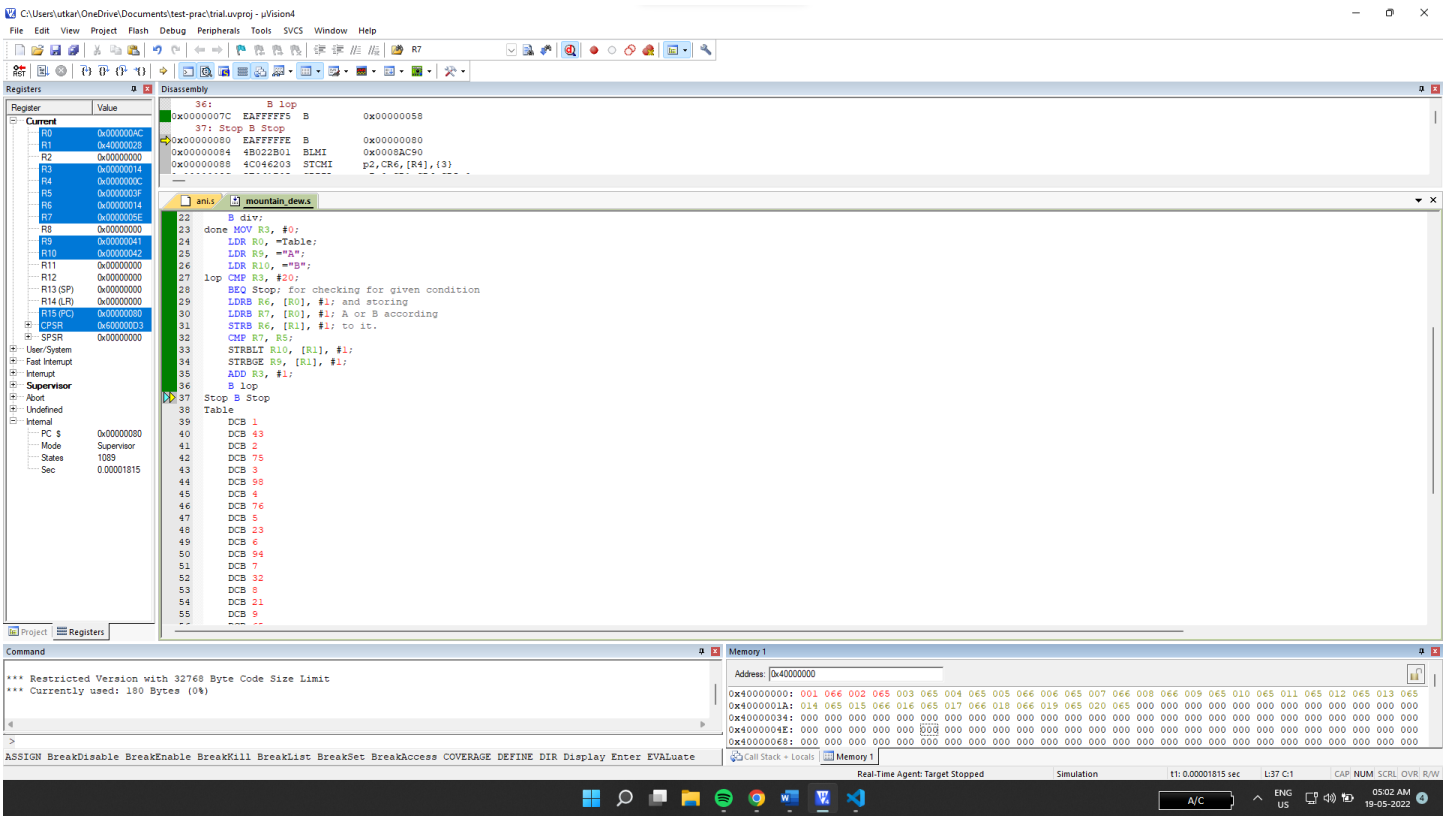
iv) If the mark is less than average put the grade “B” else put “A”. Make this a table with roll number (1byte) and grade (1byte) in the memory.

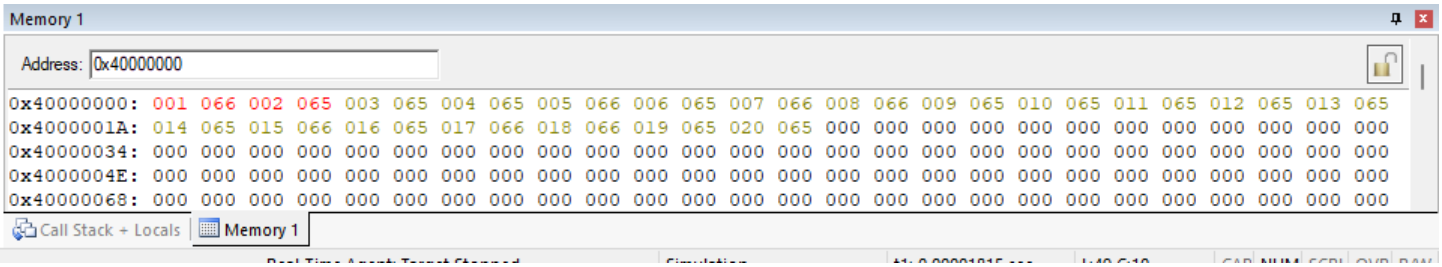
Code:

```
    AREA test,CODE, READWRITE
    EXPORT Reset_handler
Reset_handler
    LDR R0, = Table; input
    LDR R1, =0x40000000; result
    LDR R2, [R1];
    MOV R4, #0; Sum
    MOV R5, #0; AVG
    MOV R3, #0;iterator
loop CMP R3, #20;calc_sum
    BEQ div; for finding avg
    LDRB R6, [R0], #1;
    LDRB R7, [R0], #1;
    ADD R4, R4, R7;
    ADD R3, #1; increment iterator
    B loop
div CMP R4, #20;calc avg
    BLT done; dividing sum
    SUB R4, R4, #20; by 20
    ADD R5, #1;
    B div;
done MOV R3, #0;
    LDR R0, =Table;
    LDR R9, ="A";
    LDR R10, ="B";
lop CMP R3, #20;
    BEQ Stop; for checking for given condition
    LDRB R6, [R0], #1; and storing
    LDRB R7, [R0], #1; A or B according
    STRB R6, [R1], #1; to it.
    CMP R7, R5;
    STRBLT R10, [R1], #1;
    STRBGE R9, [R1], #1;
    ADD R3, #1;
    B lop
Stop B Stop
```

```
Table
    DCB 1
    DCB 43
    DCB 2
    DCB 75
    DCB 3
    DCB 98
    DCB 4
    DCB 76
    DCB 5
    DCB 23
    DCB 6
    DCB 94
    DCB 7
    DCB 32
    DCB 8
    DCB 21
    DCB 9
    DCB 65
    DCB 10
    DCB 86
    DCB 11
    DCB 73
    DCB 12
    DCB 99
    DCB 13
    DCB 94
    DCB 14
    DCB 74
    DCB 15
    DCB 38
    DCB 16
    DCB 84
    DCB 17
    DCB 30
    DCB 18
    DCB 08
    DCB 19
    DCB 65
    DCB 20
    DCB 94
    END
```

Output:





We can see the output in memory window in decimal mode. The roll numbers and their respective grades (in ascii) consecutively. Noting 65 is for A and 66 for B.