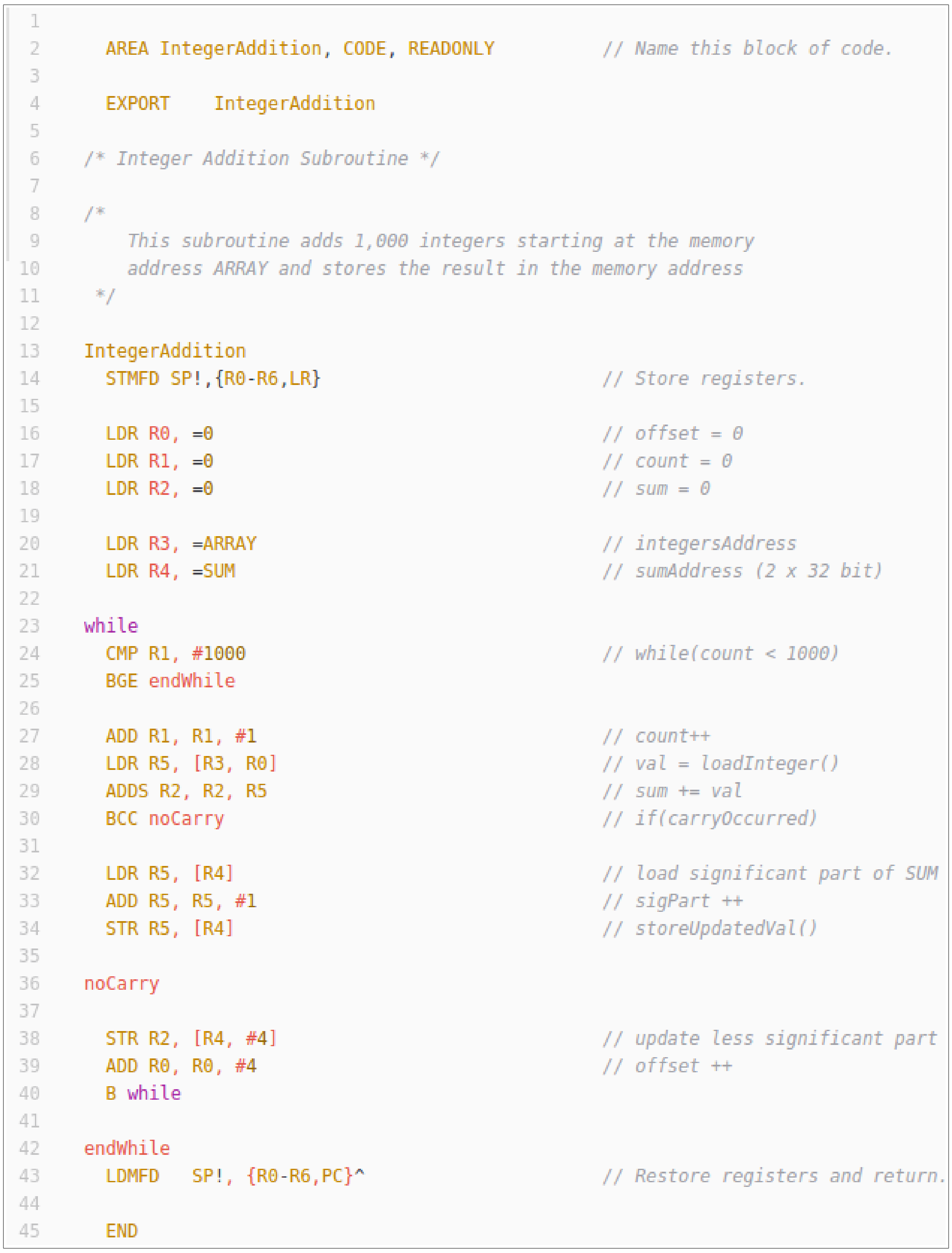
Microprocessor Exam Solutions

2016 Exam:

Q1 - Memory & Pipelining

a) Write a fragment of ARM assembly code to form the sum of 1,000 integers stored from location ARRAY upwards and to store the result at location SUM. How would you deal with arithmetic overflow.



```

  AREA IntegerAddition, CODE, READONLY          // Name this block of code.

  EXPORT    IntegerAddition

/* Integer Addition Subroutine */

/*
    This subroutine adds 1,000 integers starting at the memory
    address ARRAY and stores the result in the memory address
 */

IntegerAddition
  STMFD SP!,{R0-R6,LR}                          // Store registers.

  LDR R0, =0                                    // offset = 0
  LDR R1, =0                                    // count = 0
  LDR R2, =0                                    // sum = 0

  LDR R3, =ARRAY                                // integersAddress
  LDR R4, =SUM                                  // sumAddress (2 x 32 bit)

while
  CMP R1, #1000                                 // while(count < 1000)
  BGE endWhile

  ADD R1, R1, #1                                // count++
  LDR R5, [R3, R0]                              // val = loadInteger()
  ADDS R2, R2, R5                               // sum += val
  BCC noCarry                                   // if(carryOccurred)

  LDR R5, [R4]                                  // load significant part of SUM
  ADD R5, R5, #1                                // sigPart ++
  STR R5, [R4]                                  // storeUpdatedVal()

noCarry

  STR R2, [R4, #4]                              // update less significant part
  ADD R0, R0, #4                                // offset ++
  B while

endWhile
  LDMFD   SP!, {R0-R6,PC}^                      // Restore registers and return.

  END
```

b) Give an account of what cache is. How is cache organised and managed?

Main memory is very slow by comparison with instruction execution. It is extremely expensive and time consuming to read/write to/from memory and this has direct impacts with the performance time of a system.

As a result Cache sits logically between main memory and the CPU. It serves the purpose of implementing a middle ground between the CPU and main memory and allows for extremely fast memory accesses that can be used in a program.

There can be different levels of cache:

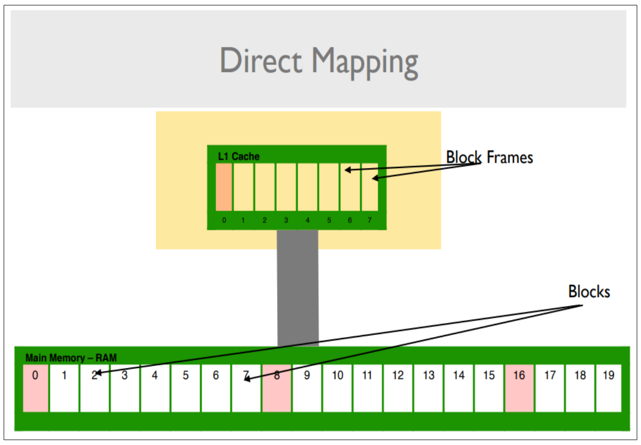
1. L1 – On Chip, Closest to the CPU, Very Fast, Very Small
2. L2 – On Chip, between CPU and Bus
3. L3 – Off Chip, between chip and main memory

Typically the L1 cache is divided into an Instruction Cache (I-Cache) and Data Cache (D-Cache)

Caches are un-named and un-numbered memory stores managed by hardware in response to run time conditions. Caches usually but not always contain duplicates of the contents of memory locations.

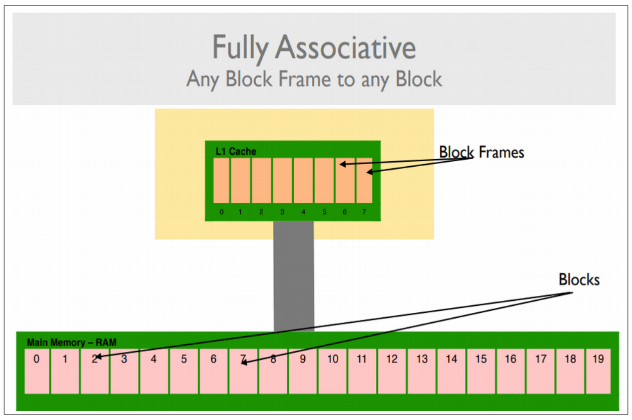
A cache is organised as a (very small) number of block frames each capable of holding a chunk of contiguous main memory locations in a structure called a 'cache line' or a 'cache block'. Due to problems building tag RAM there are different kinds of cache management

1. Direct Mapping



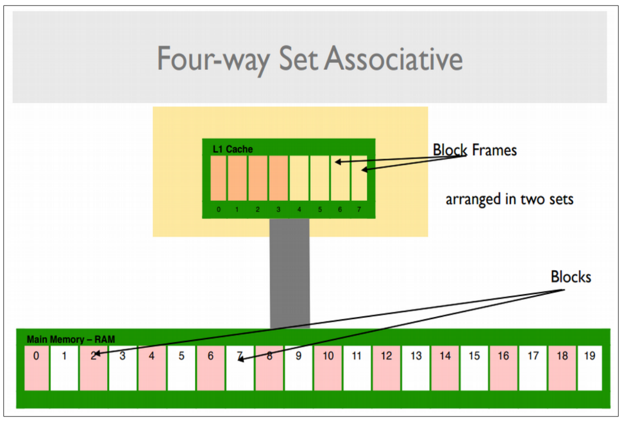
- Each block frame in cache can only cache a subset of memory blocks.
- Faster.
- Restrictive

2. Fully Associative Mapping



- Any RAM block can be mapped into any block frame in cache.
- Simple to understand.
- Problem is that to find a reference the entire tag RAM must be searched.

3. Four-Way Set Associative Mapping



- Blocks and block frames are divided into sets.
- Full associativity is possible within the sets.

Caches are invariably smaller than main memories therefore there is pressure for space within them. As programs execute over time caches will fill up and new references will have to be accomodated. This requires some form of Cache Replacement / Eviction Policy. Ideally this replacement policy should have minimum effect on the overall performance.

A Cache Miss is when a reference is made that the cache can't satisfy. There are three main reasons for a cache miss occuring:

1. Compulsory Miss: The reference was never stored in cache.
2. Collision Miss: A reference was stored in cache but was evicted because of replacement.
3. Capacity Miss: A reference was stored in cache but was evicted because of memory.

c) Assuming a three-stage pipeline, a 1 nanosecond processor clock, 10 nanosecond main memory and a cache that can be accessed by the processor instantly, estimate the amount of time it would take to execute the main loop of your code fragment above.

/ These are taken from elsewhere, not applicable to solution above*/*

Estimation of time:

- 1005 Memory Read/Writes => 10ns each = 10050ns
- 2002 Move/Add/Sub operations => 1ns each = 2002ns (if no overflow)
- At least 1000 branch operations => 3ns each (clearing pipeline) = 3000ns (if no overflow)

  =15052ns

  = 0.015052ms

Problems with this estimation:

- Determining how many times overflow would occurr
- Branch prediction difficult
- Determining what would be stored in cache during execution

Q2 - Modes & Interrupts

a) List the different modes available in the ARM processor. What are they for, and how does the ARM processor move between them? Write a fragment of code to put the processor into the user mode.

- User: Unprivileged mode, used for execution of normal programs.

  Access to : R0 – R12, SP, LR, PC and CPSR.

- FIQ: Fast Interrupt Request mode this is used when responding to interrupts from outside of the system in a quick manner. It is used when handling high priority interrupt requests.

  Access to : R8 – R14, SPSR.

- IRQ: Interrupt Request mode this is used when responding to general-purpose interrupt requests.

  Access to : R13 – R14, SPSR.

- Supervisor: Privileged mode used when OS calls (Software Interrupts – SWI's) occur within a system.

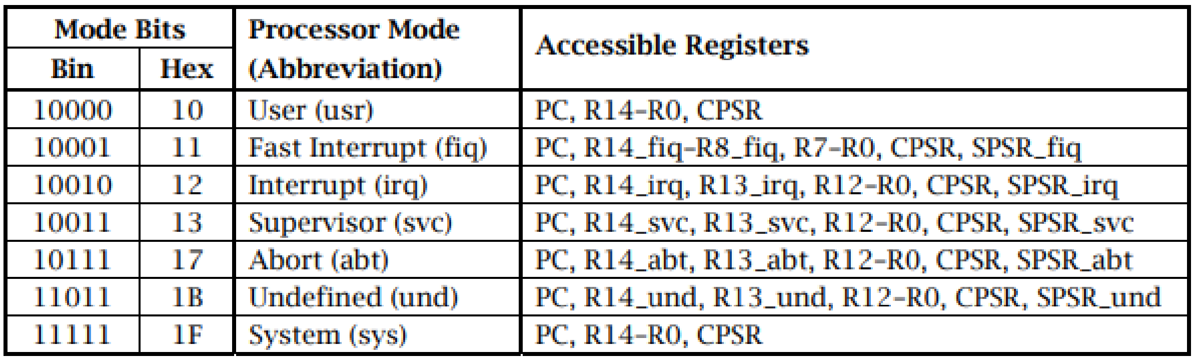
  Access to : R13 – R14, SPSR.

- Undefined: Mode used to handle undefined instructions that occur within program execution.

  Access to : R13 – R14, SPSR.

- System: Privileged mode, *but* with access to all registers. It is a cross between User and Supervisor mode offering the privileges of Supervisor with the register banks of User.

  Access to : R0 – R12, SP, LR, PC, CPSR.

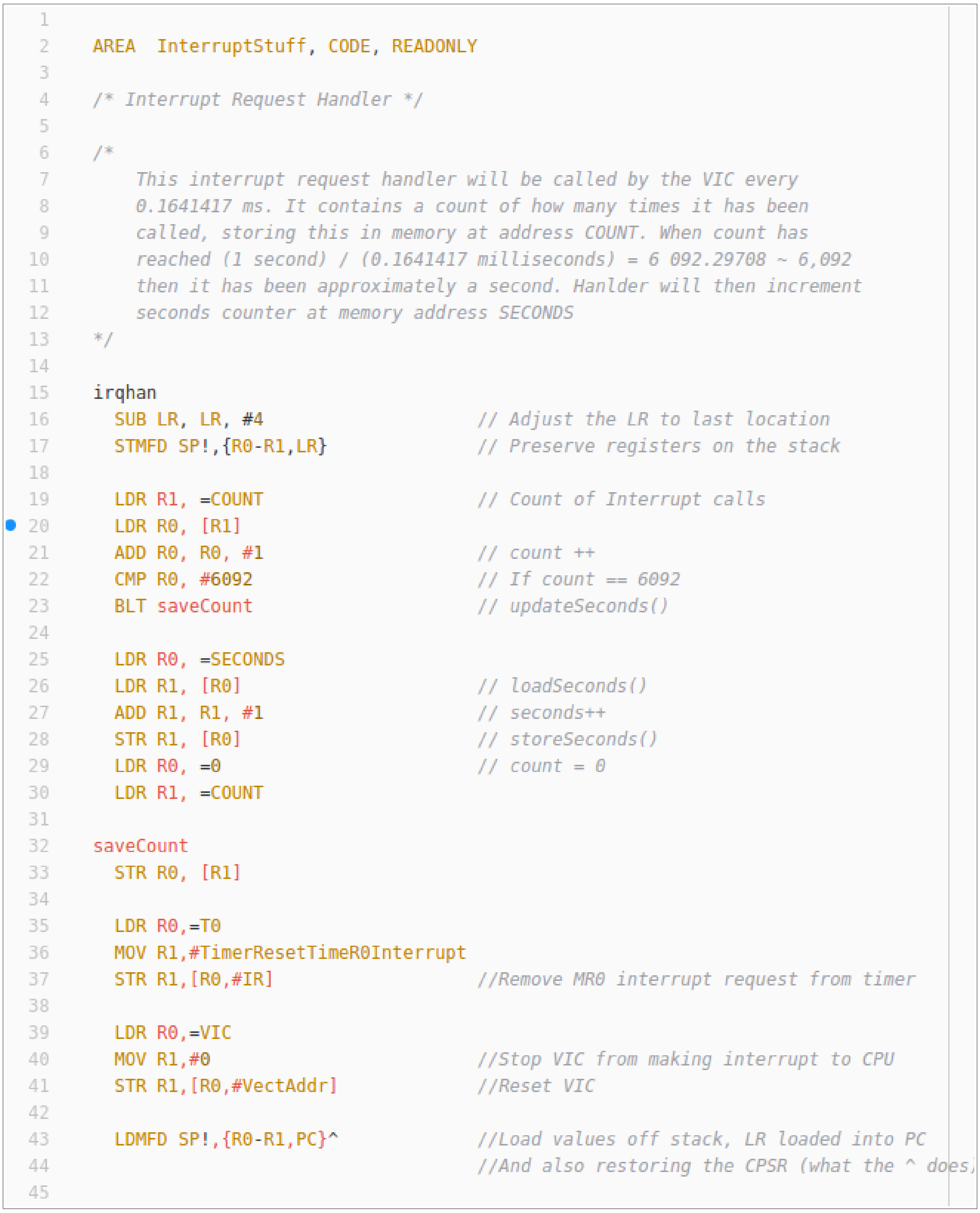


Mode Bits		**Processor Mode**	**Accessible Registers**
Bin	**Hex**	**(Abbreviation)**	
10000	10	User (usr)	PC, R14-R0, CPSR
10001	11	Fast Interrupt (fiq)	PC, R14_fiq-R8_fiq, R7-R0, CPSR, SPSR_fiq
10010	12	Interrupt (irq)	PC, R14_irq, R13_irq, R12-R0, CPSR, SPSR_irq
10011	13	Supervisor (svc)	PC, R14_svc, R13_svc, R12-R0, CPSR, SPSR_svc
10111	17	Abort (abt)	PC, R14_abt, R13_abt, R12-R0, CPSR, SPSR_abt
11011	1B	Undefined (und)	PC, R14_und, R13_und, R12-R0, CPSR, SPSR_und
11111	1F	System (sys)	PC, R14-R0, CPSR

Table 2: ARM Processor Modes

```

#define MODE_USR        0x10   /* Never use this one, as there is no way back!
#define MODE_FIQ        0x11   /* banked r8-r14 */
#define MODE_IRQ        0x12
#define MODE_SVC        0x13
#define MODE_MON        0x16
#define MODE_ABT        0x17
#define MODE_UND        0x1B
#define MODE_SYS        0x1F   /* Same as user... */

// Enter User Mode and set its Stack Pointer
      MSR   CPSR_c, #MODE_USR
      MOV   SP, R0

// Alternative solution if #MODE_USR is undefined

      MRS   R0, CPSR         // load CPSR into register
      BIC   R0, R0, #0x1F    // clear the mode field
      ORR   R0, R0, #0x10    // set bits for user mode
      MSR   CPSR_c, R0

/*
  Note that CPSR_c is used instead of CPSR in the MSR instruction,
  to avoid altering the condition code flags.
*/
```

b) Write an interrupt handler that is called by a quartz crystal-controller clock interrupt every 0.1641417 ms to maintain a seconds counter in location SECONDS. Your code must introduce no extra inaccuracies to the time by making any approximations.



```
AREA  InterruptStuff, CODE, READONLY

/* Interrupt Request Handler */

/*
    This interrupt request handler will be called by the VIC every
    0.1641417 ms. It contains a count of how many times it has been
    called, storing this in memory at address COUNT. When count has
    reached (1 second) / (0.1641417 milliseconds) = 6 092.29708 ~ 6,092
    then it has been approximately a second. Hanlder will then increment
    seconds counter at memory address SECONDS
*/

irqhan
  SUB LR, LR, #4                    // Adjust the LR to last location
  STMFD SP!,{R0-R1,LR}              // Preserve registers on the stack

  LDR R1, =COUNT                    // Count of Interrupt calls
  LDR R0, [R1]
  ADD R0, R0, #1                    // count ++
  CMP R0, #6092                     // If count == 6092
  BLT saveCount                     // updateSeconds()

  LDR R0, =SECONDS
  LDR R1, [R0]                      // loadSeconds()
  ADD R1, R1, #1                    // seconds++
  STR R1, [R0]                      // storeSeconds()
  LDR R0, =0                        // count = 0
  LDR R1, =COUNT

saveCount
  STR R0, [R1]

  LDR R0,=T0
  MOV R1,#TimerResetTimeR0Interrupt
  STR R1,[R0,#IR]                   //Remove MR0 interrupt request from timer

  LDR R0,=VIC
  MOV R1,#0                         //Stop VIC from making interrupt to CPU
  STR R1,[R0,#VectAddr]             //Reset VIC

  LDMFD SP!,{R0-R1,PC}^             //Load values off stack, LR loaded into PC
                                    //And also restoring the CPSR (what the ^ does
```

Q3 – Interrupts

a) Explain exactly what the context of a program is. How does an interrupt handler preserve the context of programs when it interrupts a program.

The context of a program can be considered in many different forms. In large it is the status of the all registers currently being used within the program. The context also encapsulates what mode the current program is in, e.g User, System, Supervisor etc. It also takes into account the value of the conditional flags at a given moment e.g V, C, N, Z. All of these details must be preserved in order for the processor to resume execution exactly as normal when the interrupt has finished.

An interrupt handler preserves the context of a program when it causes an interrupt by performing the following steps:

Preservation Steps:

1. Change into System Mode (privilege needed to return to IRQ mode)

2. Get the SP and LR of program.

3. Change into Interrupt Request Mode (IRQ)

4. Store SPSR, PC, LR, SP onto Programs Stack

5. Store Registers R0 – R12 onto Programs Stack

6. Perform Interrupt functionality

7. Return from Interrupt restoring original programs R0 – R12, PC (from LR) and CPSR.

```
LDMFD     SP!,{PC}^
```

b) Write a fragment of an interrupt handler to save the entire register and CPSR context of a user mode program.

```

  AREA  InterruptStuff, CODE, READONLY

  /* Interrupt Request Handler */

  /*
      This interrupt request handler will fully preserve the context
      of the original program it has been called from

      MRS = Move CPSR to a RX
      MSR = Move RX to CPSR
  */

  irqhan
-------------------------------------------------------------------------------
  /* 1 - Change into System Mode */
-------------------------------------------------------------------------------
    MRS R2, CPSR                // load CPSR into register
    BIC R2, R2, #0x1F           // clear the mode field
    ORR R2, R2, #0x1F           // set system mode
    MSR CPSR_c, R2              // (privilege needed to return to IRQ mode)

-------------------------------------------------------------------------------
  /* 2 - Get previous SP and LR */
-------------------------------------------------------------------------------
    MOV R0, SP                  // retrieve user mode SP
    MOV R1, LR                  // retrieve user mode LR

-------------------------------------------------------------------------------
  /* 3 - Change into IRQ Mode */
-------------------------------------------------------------------------------
    MRS R2, CPSR_c              // extract CPSR to R2
    BIC R2, R2, #0x1F           // clear the mode field
    ORR R2, R2, #0x12           // set bits for IRQ mode
    MSR CPSR_c, R2              // store new mode to CPSR

-------------------------------------------------------------------------------
  /* 4 - Store SPSR, PC, LR, SP onto Program Stack */
-------------------------------------------------------------------------------
    MRS R2, SPSR                // extract SPSR to R2
    STR R2, [R0, #4]            // store the spsr(CPSR) of program to SP

    STR LR, [R0, #8]            // store PC
    STR R1, [R0, #12]           // store LR
    STR R0, [R0, #16]           // store SP
    ADD R0, #16                 // update stack pointer to "true" value

-------------------------------------------------------------------------------
  /* 5 - Store Registers R3 - R12 (Unchanged) onto Program Stack */
-------------------------------------------------------------------------------
    STMFD R0!, {R3-R12}         // store register contents on the stack

```

```
-------------------------------------------------------------------------------
 /* 6 - Get original R0 - R2 values and push onto stack */
-------------------------------------------------------------------------------
    LDMFD SP!, {R3-R5}          // take R0-R2 off of IRQ stack
    STMFD R0!, {R3-R5}          // store onto programs stack

    MOV R0, R3                  // move register contents back to original pos
    MOV R1, R4                  // overwriting LR of user mode
    MOV R2, R5;                 // overwriting SP of user mode

    LDMFD SP!, {R3-R5}          // restore registers to condition before context
                                // storing began
                                // * system context storing done #*

      //clear the interrupt here, and perhaps other Irq functionality

    LDMFD SP!,{PC}^             // return from interrupt, restoring pc from lr
                                // and also restoring the CPSR

```