Important Questions

Level 2

Q. List & explain the basic 'C' Data Types

A:- C compiler datatype mappings

C Data Type	Implementation
char	unsigned 8-bit byte
short	signed 16-bit halfword
int	signed 32-bit word
long	signed 32-bit word
long long	signed 64-bit double word

Q. List & explain the following instructions of ARM with proper syntax & example for each.

A:- I) Branch Instructions

1.	B	branch	pc = label (program counter)
2.	BL	branch with link	pc = label lr = address of the next instruction after the BL (link register)

A branch instruction changes the flow of execution or is used to call a routine. This type of instruction allows programs to have subroutines, if-then-else structures, and loops.

(Just for your understanding:
r14 -> lr : where the ARM processor puts the return address whenever it calls a subroutine.
r15 -> pc : contains the address of the next instruction to be fetched by the processor.)

Syntax :- B {<cond>} label

BL {<cond>} label

Eg:- a) To change flow of execution

```
    B   forward
    ADD r1, r0, #4
forward
    SUB r1, r0, #4
```

Eg:- 5) To call a subroutine

```
BL      subroutine      ; branch to subroutine
CMP     r1, #5
MOVEQ   r1, #0
:
subroutine
    <subroutine code>
    MOV   pc, lr        ; return by moving pc=lr
```

II) LOAD-STORE instructions.

Load-store instructions transfer data between memory & processor registers.

Syntax:- <LDR|STR>{<cond>}{B} Rd, addressing^1

LDR{<cond>}SB|H|SH Rd, addressing^2

STR{<cond>}H Rd, addressing^2

Load store instructions

1. LDR : Load word (32-bit)
2. LDRH : Load half-word (16-bit)
3. LDRB : Load a byte (8-bits)

} into a Register.

4. STR : store 32-bit
5. STRH : store 16-bit
6. STRB : store 8-bit

} from a Register.

7. LDRSB :- load signed byte (8-bits) into a Register.
8. LDRSH :- load signed halfword into a Register.

eg:-

```
;
; load register r0 with the contents of the memory address
; pointed to by register r1.
LDR   r0, [r1]
; store the contents of register r0 to the memory address
; pointed to by register r1.
STR   r0, [r1]
```

III) SWAP INSTRUCTION

- The swap instruction swaps the contents of memory with the contents of a register.

Syntax:- SWP{B}{<cond>} Rd, Rm, [Rn]

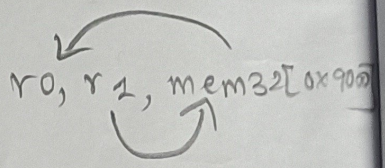
swp	swap a word between memory and a register	tmp = mem32[Rn] mem32[Rn] = Rm Rd = tmp
SWPB	swap a byte between memory and a register	tmp = mem8[Rn] mem8[Rn] = Rm Rd = tmp

eg:- PRE:

mem32[0x9000] = 0x12345678

r0 = 0x00000000

r1 = 0x11112222



POST:

mem32[0x9000] = 0x11112222

r0 = 0x12345678

r1 = 0x11112222

IV) PROGRAM STATUS REGISTER INSTRUCTIONS

The ARM instruction set provides two instructions to directly control a program status register (psr):

a) MRS - MOV into Register from Status

It transfers contents of either CPSR or SPSR into a register.

b) MSR - MOV into Status from Register

It transfers contents of register to CPSR or SPSR

Syntax:-

MRS{<cond>} Rd, <cpsr|spsr>

MSR{<cond>} <cpsr|spsr>_<fields>, Rm

MSR{<cond>} <cpsr|spsr>_<fields>, #immediate

Eg:-

```
PRE   cpsr = nzcvqiFt_SVC

      MRS   r1, cpsr          ; copies cpsr into register r1
      BIC   r1, r1, #0x80     ; 0b01000000   ; The BIC instruction clears bit 7 of r1

      MSR   cpsr_c, r1

POST  cpsr = nzcvqiFt_SVC
```

Q. Develop an ALP to illustrate the working of SWAP instruction

Ans:- (Pgm1)

```
        AREA  swapMC, CODE, READONLY
              ENTRY
              LDR  R2, =BLOCK1
              LDR  R3, =BLOCK2
              LDR  R4, [R2]
              STR  R4, [R3]
              MOV  R0, #0x00000000
              MOV  R1, #0x10000002  ; Immediate value I've taken
              SWP  R0, R1, [R3]
              B    L
L
BLOCK1  DCD   0x12345678   ; Declaring the data
        AREA  MYDATA, DATA, READWRITE
BLOCK2  DCD   0
        END
```

Pgm 2 (Swap Byte)

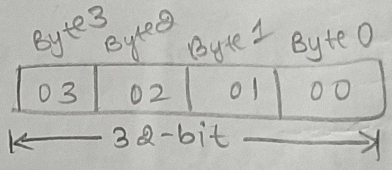
```
        AREA    swapMC, CODE, READONLY
        ENTRY
        LDR     R2, =BLOCK1
        LDR     R3, =BLOCK2
        LDRB    R4, [R2]    ; loading 8-bit data.   1byte=8bit
        STRB    R4, [R3]    ; storing 8-bit data

        MOV     R0, #00
        MOV     R1, # 0X05  ; Immediate value I've taken
        SWPB    R0, R1, [R3]

L       B       L
BLOCK1  DCB     0X12        -> It is 8-bit data. But it is hexadecimal
        AREA    MYDATA, DATA, READWRITE
BLOCK2  DCB     0
        END
```

Q.8. Develop an ALP to illustrate the working of Load store instructions:

Ans:-




```
        AREA    LDRpgm, CODE, READONLY
        ENTRY
        MOV     R4, #4
        LDR     R0, =FBLOCK
        LDR     R1, =SBLOCK
LOOP    LDR     R3, [R0], #4 ; we are loading complete 32-bit
                               data into the register from
                               memory.
        STR     R3, [R1], #4 ; we are storing register value into
                               memory.
        SUBS    R4, #1  ; subtract 1 from R4, updating flags
        BNE     LOOP    ; Branch to loop if result is not
                          zero. (ie. if Zflag ≠ 1 then branch
                          to loop.)
FBLOCK  DCD     0X12345678, 0X9ABCDEF0,
                0X00004000, 0X0000C000
L       B       L
        AREA    MYDATA, DATA, READWRITE
SBLOCK  DCD     0
        END
```

we index it by 4 as we are reading complete 32-bit data

If result=0, then Zflag = 1.

Level 4 :- Analyze

② Analyze the given piece of codes ('c' code & compiler (v1) output) and answer the following :

→ • what is the drawback of using char datatype for declaring the local variables in ARM 7 'C' program?

Ans: If we make use of char datatype to declare the local variables then we must make use of additional instruction AND to fetch 8-bit data.
(That is AND r1, r1, #0xFF)

• In the compiler output how can we avoid the instruction AND R1, R1, #0xFF?

Ans:- If you declare the local variable by int then you can remove the additional instruction AND.

• what is the use of BCC instruction?

Ans:- BCC (Branch if Carry is Cleared) i.e. If carry is cleared then we'll loop it.

• Why PC is updated with R14 content? can we replace R14 by any other register?

Ans:- r14 (ie. lr) : where the ARM processor puts the return address whenever it calls a subroutine

r15 (i.e. pc) : contains the address of the next instruction to be fetched by the process.

So, MOV pc, r14 is copying the return address from the r14 back to the PC.

You can write it as Mov pc, lr also.

v2
(3)

- Which data type is used to declare the local variable?

Ans:- int is the datatype used to declare the local variable.

- What is the modification that is required in the compiler output if the variable sum is 16-BIT.

Ans:- If I want to extract 16-bit data, I can use short datatype to declare the local variable sum in C program. And accordingly, in the Assembly code (ie the compiler output) three additional instructions must be added.

v3 →

1. ADD instruction: as I cannot perform barrel shifting operations on LDRH
2. MOV instruction: to remove/vanish upper 16-bit data from 32-bit data, I do LSL #16
3. MOV instruction: when I do LSR, my signed bit will be lost, so, I make use of ASR so that I'll retain the signed bit.

Ans:- (You can write as)

The cast reducing total + array[i] to a short (16-bit data) requires two MOV instructions.

The compiler shifts left by 16 & then Right by 16 to implement, a 16-bit sign extend.

(4) Analyze the given piece of codes ('c' & ARM compiler output) & answer the following.

- How can we reduce these instructions in the compiler output?

```
ADD   r3, r2, r1, LSL #1
```

```
MOV   r0, r0, LSL #16
MOV   r0, r0, ASR #16
```

Ans :- a) We can reduce the first instruction (ie ADD) by accessing the array by incrementing the pointer data rather than using an index as in data[i]. This is efficient regardless of array type size or element size. All ARM load & store instructions have a post increment addressing mode

b) You can reduce the other two MOV instructions by using an int type variable to hold the partial sum. We only reduce the sum to a short type at the function exit.

- Rewrite the 'C' code to reduce these instructions

Ans The checksum_v4 code fixes all the problems faced in checksum_v3. It removes three instructions from the inside loop, saving three cycles per loop compared to checksum_v3.

```
checksum_v4
        MOV   r2, #0      ; sum = 0
        MOV   r1, #0      ; i = 0
checksum_v4_loop
        LDRSH r3, [r0], #2    ; r3 = *(data++)
        ADD   r1, r1, #1      ; i++
        CMP   r1, #0x40       ; Compare i, 64
        ADD   r2, r3, r2      ; sum += r3
        BCC   checksum_v4_loop ; if (sum<64) goto loop
        MOV   r0, r2, LSL #16
        MOV   r0, r0, ASR #16  ; r0 = (short) sum
        MOV   pc, r14          ; return r0
```

UNIT 3:

Level 2:

1. List & explain the various registers of ARM7 used for configuring ports as GPIO, INPUT/OUTPUT and SET/CLEAR.

Ans:- Here are some key registers used for configuring ports in ARM7:

(a) PINSEL (Pin Function Select Register):

This register selects the function of each pin on the microcontroller. Bits in this register determine whether a pin will function as a GPIO, as an alternate function, or other functionalities.

PINSEL 0 } used to configure port 0.
PINSEL 1 }
PINSEL 2 → used to configure port 1 ($P_{1.0} - P_{1.15}$)

* PINSEL 0 used to access pins from $P_{0.0}$ to $P_{0.15}$
* PINSEL 1 " " " " " $P_{0.16}$ to $P_{0.31}$
* PINSEL 2 " " " " " $P_{1.0}$ to $P_{1.15}$ (only 16 pins are available to the programmer)

* The combination 00 is always for GPIO functionality.

(b) IO×DIR (GPIO Port Direction control register):

This is a 32-bit wide register. It individually controls the direction of each port pin. Setting a bit to '1' configures the corresponding pin as an output pin. Setting a bit to '0' configures the corresponding pin as an Input pin.

1 → Output
0 → Input

(c) IO×SET (GPIO Port Output Set register):

→ 32-bit wide register.
→ Used to make pins of PORT (PORT 0 / PORT 1) HIGH.
→ Writing one to specific bit makes that pin HIGH.
Writing zero has no effect.

(d) IOxCLR (GPIO Port Output Clear register):

This is
→ a 32-bit wide register.

→ Used to make pins of port low.
→ writing one to specific bit makes that pin low.
→ writing zeroes has no effect.

• To configure port

Q. What value has to be loaded into the Registers:

• To configure port 0 (P0.0 - P0.15) pins as input?

Ans :- 'zero' for input..

```
PINSEL0 = 0x00000000; // configure P0.0 - P0.15 as GPIO
IO0DIR = 0x00000000; // configure P0.0 - P0.15 as INPUT
```

Q. • To configure Port0 (P0.16 - P0.31) pins as input?

Ans :- 'zero' for input

```
PINSEL1 = 0x00000000; // configure P0.16 - P0.31 as GPIO
IO0DIR = 0x0000 0000; // configure P0.16 - P0.31 as INPUT
```

• To configure port0 (P0.0 - P0.15) pins as OUTPUT?

Ans :- 'one' for output..

```
PINSEL0 = 0x00000000; // configure P0.0 - P0.15 as GPIO
IO0DIR = 0xFFFFFFFF; // configure P0.0 - P0.15 as OUTPUT.
```

• To configure PORT0 (P0.16 - P0.31) pins as OUTPUT?

Ans :- 'one' for output.

```
PINSEL1 = 0x00000000; // configure P0.16 - P0.31 as GPIO
IO0DIR = 0xFFFFFFFF; // configure P0.16 - P0.31 as OUTPUT
```

• To configure PORT1 ($P_{1.16}$ - $P_{1.31}$) pins as INPUT?

Ans:- 'zero' for input.

```
PINSEL2 = 0X00000000; // configure P1.16-P1.31 as GPIO
IO1DIR = 0X00000000; // configure P1.16-P1.31 as INPUT
```

• To configure PORT1 ($P_{1.16}$-$P_{1.31}$) pins as OUTPUT?

Ans:-

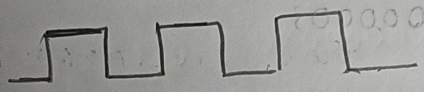
```
PINSEL2 = 0X00000000; // configure P1.16-P1.31 as GPIO
IO1DIR = 0XFFFFFFFF; // configure P1.16-P1.31 as OUTPUT.
```

Level 3 :-

1. Develop an embedded 'c' program to blink the LEDs connected to Port 0 pins (P0.16 - P0.23).

Ans:-
```
#include<LPC21xx.h>
unsigned int delay;
int main()
{
  PINSEL1 = 0X00000000; //configure P0.16 to P0.23 as GPIO
  IO0DIR = 0XFFFFFFFF; //configure P0.16 to P0.23 as OUTPUT
  while(1)
  {
    IO0SET = 0X00FF0000; //SET PINS 16-23 of PORT 0   (pin no 16-23)
    for(delay=0; delay<1000; delay++);
    IO0CLR = 0X00FF0000; //CLEAR PINS 16-23 of PORT 0
    for(delay=0; delay<10000; delay++);
  }
}
```

Output :-



2. Develop an embedded 'c' program to implement 8-bit binary counter on port 0 pins (P0.16 - P0.23).

Ans:-
```
#include<LPC21xx.h>   --> P0.16 - P0.31 ahi program it is
void delay(void);
unsigned int count;
int main()
{
  unsigned int comp = 0; //compliment
  PINSEL1 = 0X00000000; //configure PORT0 (16-31) as GPIO
  IO0DIR = 0XFFFFFFFF;
  while(1){
    for(count = 0; count<=0XFF; count++){
      comp = (~count); //just to ensure that after 255, 0 should come.
      comp = comp & 0X000000FF; // to fetch lower 8-bit data
      IO0PIN = (comp<<16); //I'm sending it on I/O pin. we can see the status of count from this pin. IO0PIN bez I'm using port 0.
      delay();
    }
  }
}
```

```
void delay(void)
{
  unsigned int i;
  for(i=0 ; i<6500000 ; i++);
}
```

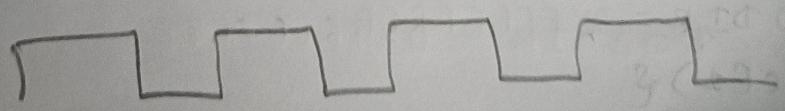
Q. Develop an embedded 'c' program to interface DAC with ARM7 to generate the following waveforms.

a) SQUARE wave.

Ans:-

```
#include <LPC21xx.h>
void delay(void);
int main()
{
  PINSEL0 = 0X00000000; //P0.10 - P0.15 as GPIO
  PINSEL1 = 0X00000000; //P0.16 - P0.31 as GPIO
  IO0DIR = 0XFFFFFFFF; //P0.0 - P0.31 configured as
                       OUTPUT.
  while(1)
  {
    IO0PIN = 0X00000000;
    delay();
    IO0PIN = 0XFFFFFFFF;
    delay();
  }
}
void delay()
{
  unsigned int i;
  for(i=0; i<500; i++);
}
```

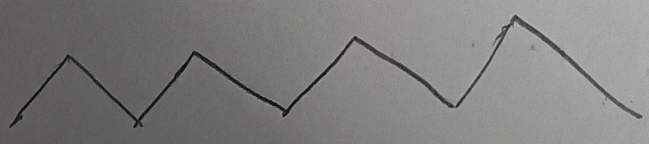
Output:-



b) TRIANGULAR WAVE

```
#include<LPC21xx.h>
int main()
{
  unsigned long int temp= 0x00000000;
  unsigned int i=0;
  // By default all pins are configured as GPIO
  IODIR = 0xFFFFFFFF;  // P0.0-P0.31 as OUTPUT
  while(1)
  {
    for(i=0 ; i!=0xff; i++)
    { temp = i; // I'm getting 16-bit data on a 32-bit register,
                   with upper 16-bit data padded with zeroes.
      // content of temp must be shifted left by 16 times to bring
       the lower order data to higher order data. (as I don't want
       the padded zero's)
      temp = temp<< 16;
      IOPIN = temp;
    }
    for(i=0xff; i!=0; i--)
    {
      temp = i;
      temp = temp << 16;
      IOPIN = temp;
    }
  }
}
```

Output:-

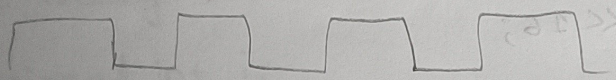


Q.4. Develop an Embedded 'C' program to interface the Relay with ARM7.

Ans—

```
#include <LPC21xx.h>
unsigned int i;
int main()
{
  // I'm not writing PINSEL0 and PINSEL1 because
  By default PORT0 and PORT1 pins are configured
  as GPIO.
  // the relay pin is connected to pin no.10 of port0.
IO0DIR = 0x00000400; // Set P0.10 as output.
IO0SET = 0x00000400; // P0.10 is set to high
while(1)
{
   for(i=0; i<1000000; i++);
   IO0SET = 0x00000400; // Relay ON
   for(i=0; i<100000; i++);
   IO0CLR = 0x00000400; // Relay OFF // means the relay will be turned off as we cleared it.
}
}
```

output—



5Q. Develop an embedded 'c' program to blink the Built in LED connected to pin number 5 of arduino UNO.

Ans:- Program :-

```
void setup(){
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);
}
void loop(){
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on
  Serial.println("LED on");
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW); // turn the LED off
  Serial.println("LED off");
  delay(1000);
}
```

6.Q. Develop an embedded 'c' program to interface LDR sensor connected to pin number 13 of Arduino UNO.

Ans:-

```
int light_pin =13 ;
void setup(){
  pinmode(light_pin, INPUT);
  Serial.begin(9600);
}
void loop(){
  int light_data = digitalRead(light_pin);
  if(light_data)
    Serial.println("Light Not Detected!");
  else
    Serial.println("Light Detected");
  delay(1000);
}
```

Develop an embedded 'c' program to interface buzzer connected to pin number 9 of arduino UNO

```
// RM1 - RM9 connected
int buzzer_pin = 9; // Arduino pin
void setup() {
    pinMode(buzzer_pin, OUTPUT);
    Serial.begin(9600);
    digitalWrite(buzzer_pin, HIGH);
}
void loop() {
    digitalWrite(buzzer_pin, LOW);
    delay(1000);
    digitalWrite(buzzer_pin, HIGH);
    delay(1000);
}
```

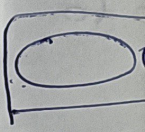
TW6 (on Sorting)

Question:-

Program:-

```
        AREA MCTW6, CODE, READONLY
        ENTRY
        MOV R8, #4 ; Counter register for array
        LDR R1, =CVALUE ; R1 points to the array present in CODE region
        LDR R2, =DVALUE ; R2 points to the array present in
                          DATA region
LOOP0   LDR R3, [R1], #4 ; Fetch a number from first array
        STR R3, [R2], #4 ; paste it in 2nd array
        SUBS R8, R8, #1 ; Decrement counter register
        CMP R8, #0      ; Compare R8 with zero
        BNE LOOP0       ; Once R8 becomes zero it goes out of LOOP0:
                          If R8!=0, you continue data transfer
                          Once R8=0, come out of loop.
START   MOV R7, #0      ; Swap flag is cleared. ~~If there are~~
                        ; If there are n-elements, then number
                          of comparisons = n-1. Let me take one
                          register (say R5) to hold the no. of
                          comparisons.
        MOV R5, #3      ; Comparison COUNT.
        LDR R2, =DVALUE
LOOP2   LDR R1, [R2], #4 ; means LDR R1, content of R2 ; 1st number in R1
        LDR R3, [R2]     ; 2nd number in R3
        CMP R1, R3 ; R1-R3 will happen, then make use of branch instruction
        BLT LOOP1 ; BLT means Branch if Lesser Than. If you
                    don't want to exchange (i.e., swap) then
                    branch to LOOP1.
```

; else if you want to swap the two numbers then perform the following 4 steps. In these 4, the 1st two steps actually perform swapping.



LOOP1

L

CVALUE

DVALUE

Regis

Regi

R0
R1
R2
R3
R4
R5
R6
R7

R1

```
        STR  R1, [R2], #-4
        STR  R3, [R2]
        MOV  R7, #1 ; SWAP = 1 ; flag to indicate that swapping has occured.
        ADD  R2, #4 ; Restore Pointer
LOOP1   SUBS R5, R5, #1
        CMP  R5, #0
        BNE  LOOP2
        CMP  R7, #0
        BNE  START
L       B    L
CVALUE  DCD  0X44444444, 0X11111111, 0X33333333, 0X22222222
        AREA MYDATA, DATA, READONLY
DVALUE  DCD  0
        END
```

Output

Registers

Register	Value
R0	0X00000000
R1	0X33333333
R2	0X4000000C
R3	0X44444444
R4	0X00000000
R5	0X00000000
R6	0X00000000
R7	0X00000000
⋮	
~~R15 (PC)~~	
R15 (PC)	0X00000054

Memory1

Address: 0X00000064

0X00000064: 44 44 44 44 11 11 11 11 33 33 33 33 22 22 22 22 64 00 00 00 00 00 00 40

Address: 0X40000000

0X40000000: 11 11 11 11 22 22 22 22 33 33 33 33 44 44 44 44